

AD-A078 055

MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 6/4
DEPENDENCY DIRECTED REASONING FOR COMPLEX PROGRAM UNDERSTANDING--ETC(U)
APR 79 H E SHROBE
N00014-75-C-0643

UNCLASSIFIED

AI-TR-503

NL

1 OF 4
AD-A078055



UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR-503	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Dependency Directed Reasoning for Complex Program Understanding	5. TYPE OF REPORT & PERIOD COVERED technical report	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Howard Elliot/Shrobe	8. CONTRACT OR GRANT NUMBER(s) N00014-75-C-0643	9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
10. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139	11. REPORT DATE Apr 1979	12. NUMBER OF PAGES 293
13. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209	14. SECURITY CLASS (of this report) UNCLASSIFIED	15. DECLASSIFICATION/DOWNGRADING SCHEDULE
16. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217	17. DISTRIBUTION STATEMENT (of this Report) Distribution of this document is unlimited. 14 AI-TR-503	
18. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)		
19. SUPPLEMENTARY NOTES None		
20. KEY WORDS (Continue on reverse side if necessary and identify by block number) <div style="display: flex; justify-content: space-between;"> <div> Program Analysis Program Understanding Program Verification Programmer's Apprentice </div> <div> Automated Deduction Dependency Networks Truth Maintenance </div> </div>		
21. ABSTRACT (Continue on reverse side if necessary and identify by block number) Artificial Intelligence research involves the creation of extremely complex programs which must possess the capability to introspect, learn, and improve their expertise. Any truly intelligent program must be able to create procedures, and to modify them as it gathers information from its expertise. (Sussman, 1975) produced such a system for a "mini-world"; but truly intelligent programs must be considerably more complex. A crucial stepping stone in AI research is the development of a system which can understand complex programs		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

407 483

Lm

Block 20 continued:

well enough to modify them.

There is also a complexity barrier in the world of commercial software which is making the cost of software production and maintenance prohibitive. Here too a system which is capable of understanding complex programs is a necessary step. The Programmer's Apprentice Project (Rich and Shrobe, 76) is attempting to develop an interactive programming tool which will help expert programmers deal with the complexity involved in engineering a large software system.

This report describes REASON, the deductive component of the programmer's apprentice. REASON is intended to help programmers in the process of evolutionary program design. REASON utilizes the engineering techniques of modelling, decomposition, and analysis by inspection to determine how modules interact to achieve the desired overall behavior of a program. REASON coordinates its various sources of knowledge by using a dependency-directed structure which records the justification for each deduction it makes. Once a program has been analyzed, these justifications can be summarized into a teleological structure called a plan which helps the system understand the impact of a proposed program modification.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-75-C-0643.

Accession For	
NTIS GRA&I	<input checked="checked" type="checkbox"/>
DEC TAB	<input type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By _____	
Distribution/ _____	
Availability Codes	
Dist	Availand/or special
A	

Dependency Directed Reasoning
For
Complex Program Understanding

by
Howard Elliot Shrobe

Massachusetts Institute of Technology
April 1979

Revised version of a dissertation submitted on August 31, 1978 to the Department of Electrical Engineering and Computer Science in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Abstract

Artificial Intelligence research involves the creation of extremely complex programs which must possess the capability to introspect, learn, and improve their expertise. Any truly intelligent program must be able to create procedures and to modify them as it gathers information from its experience. [Sussman, 1975] produced such a system for a "mini-world"; but truly intelligent programs must be considerably more complex. A crucial stepping stone in AI research is the development of a system which can understand complex programs well enough to modify them.

There is also a complexity barrier in the world of commercial software which is making the cost of software production and maintenance prohibitive. Here too a system which is capable of understanding complex programs is a necessary step. The Programmer's Apprentice Project [Rich and Shrobe, 76] is attempting to develop an interactive programming tool which will help expert programmers deal with the complexity involved in engineering a large software system.

This report describes REASON, the deductive component of the programmer's apprentice. REASON is intended to help expert programmers in the process of *evolutionary* program design. REASON utilizes the engineering techniques of modelling, decomposition, and analysis by inspection to determine how modules interact to achieve the desired overall behavior of a program. REASON coordinates its various sources of knowledge by using a dependency-directed structure which records the justification for each deduction it makes. Once a program has been analyzed these justifications can be summarized into a teleological structure called a *plan* which helps the system understand the impact of a proposed program modification.

Acknowledgements

I would have found this research impossible to conduct had it not been for the encouragement and cooperation given me by my supervisor Gerald Jay Sussman, my readers (and mentors) Carl Hewitt and Marvin Minsky and my close colleagues Johan deKleer, Jon Doyle, Mark Miller, Charles Rich, and Richard Waters. But most of all I would never have finished this work if it hadn't been for Annie who set up one house while the other almost came tumbling down.

CONTENTS

1. The Importance of Program Understanding	1
1.1 The Problem of Program Maintenance	4
1.2 An Imagined Scenario	7
1.3 The Research Content of This Thesis	14
2. An Engineering Theory of Evolutionary Design	17
2.1 Type of Programs -- Algorithms vs. Systems	18
2.2 What Characterizes Evolutionary Change?	20
2.3 Why Is Evolutionary Design Necessary?	22
2.4 What Do Engineers Do?	26
2.5 Plans and Teleology	30
2.6 Representing Plans	31
2.7 Plans in Maintenance and Explanation	45
2.8 Dependency Directed Reasoning	47
3. The Reasoning System	49
3.1 Dependencies and Justifications	49
4. Explicit Control and The Task Network	57
4.1 Hypotheticals	66
4.2 The Rules Of Inference	73
4.3 Closing The Reflexive Loop	85
4.4 Equality, Reference and Anonymous Objects	86
4.5 Situational Logic	92

5. Describing Programs	96
5.1 Specs - I/O Descriptions	96
5.2 Plan Diagrams	99
6. A Symbolic Interpreter for Plan Diagrams	106
7. An Example of Symbolic Interpretation	120
8. The Temporal Viewpoint	137
8.1 A Paradigmatic Example	137
8.2 Situations and Orderings	144
8.3 Temporal Collections	146
8.4 Temporal Collections Inputs and Outputs	148
8.5 Temporal Collection Data-flows	152
9. The Recognition Paradigm	156
9.1 Abstract Flows, Data and Control Pathways	162
9.2 Summary	168
10. Description of Data-Structures	169
10.1 The Data-Description Language	170
10.2 Parameterized Object Descriptions	179
10.3 Implementation and Virtual Objects	182
10.4 A Catalogue of Object Descriptions	187
11. Reasoning About Side-effects	201
11.1 Specifying Side-effects	203
11.2 Reasoning About Simple Side Effects	206
11.3 Safe-from and Not Safe-from	214
11.4 More Complicated Effects	216
11.5 Determining What's Affected	220
11.6 An Example	227
11.7 Pseudo Parallelism	231

12. Reducing Complexity in Side Effect Analysis	235
13. Reasoning About Program Modifications	241
13.1 Updating The Recognition Map	245
14. Conclusions	254
14.1 Good Decisions	254
14.2 Problems	257
14.3 Future Directions	259
15. A Survey of Related Work	261
15.1 Newer Areas of Verification Research	267
15.2 Apprentice-Like Systems	271
15.3 Dependency Based Reasoning	274
16. Bibliography	276

Chapter 1: The Importance of Program Understanding

There is a fundamental distinction between running a program to do something and asking it to understand how it accomplishes that very same task. Even huge programs, like MACSYMA [Macsyma 1975], lack the ability to introspect, to examine their own procedures. But, lacking the ability to introspect, they also lack the ability to describe their own behavior, to modify their behavior, or to rationally plan the allocation of internal resources among competing tasks. To do such tasks, a program must *understand itself*.

To understand itself a program must be able to understand programs. Artificial Intelligence programs are large and complex; they maintain large knowledge-bases, use multiple layers of interpreters, and frequently employ advanced control structures. This thesis attempts to formalize and represent some of the knowledge necessary to understand and explain such programs. It is incomplete and exploratory; more questions are raised than answered. However, I believe that several important advances are made, and hope that this work may serve as a bridge from past exploratory work of [Minsky, 1968], [McCarthy, 1968] and [Sussman, 1975] to future work on truly self-conscious systems such as proposed in [Doyle, 1978].

How can a program understand another program? In a step towards this, Sussman [Sussman, 1975] introduced a paradigm of how an intelligent computer program can acquire new skills. In this paradigm, a planning program first attempts to combine old ones procedures to form a "first order" approximation to a desired complex goal. This new procedure is then executed in a "careful mode", maintaining a record of the process. If one of several pre-defined kinds of "bugs" is recognized, the system attempts to analyze the bug and to debug its procedure so as to achieve the desired goal.

Why is it not enough for a program just to understand its subject matter? What makes us want to have it "understand itself" as well? Even among the earliest works of AI, the issue of self-consciousness was raised. McCarthy's proposal for the "Advice Taker" [McCarthy, 1968] was in essence a proposal to develop a system which understands its procedures well enough to be told how to employ them effectively, i.e. to take advice. Although many of McCarthy's original plans fell short of the mark there were many seminal ideas present as well. Similarly Minsky's Matter, Mind and Models [Minsky, 1968] raised many of these issues. Sussman's Hacker was the first substantially detailed system to exhibit a serious approach to self-conscious acquisition

2 The Importance of Program Understanding

of knowledge.

HACKER could be considered introspective in a limited sense. It knew what it had done, why it had done it, and what higher level goal each action was intended to achieve. Finally, it could examine the procedure it had coded and modify this procedure's code.

But HACKER was very limited in its expertise. It knew only about the blocks world, a miniworld with a one armed robot, a collection of blocks, and simple goals like building a tower. Such simplicity is crucial in the initial stages of scientific exploration; and HACKER remains an important milestone. However, to progress we must be capable of engineering a system which can understand procedures of complexity greater than those of HACKER. We need a system with greater expertise about programs!

Follow-up work, unfortunately, has not developed very far in this direction, until recently. Goldstein [Goldstein, 1974], in his MYCROFT system attempted to present a more sophisticated taxonomy of procedures and bugs, but he also worked in a mini-world, the domain of graphics programs written by elementary school children. Sacerdoti [Sacerdoti, 1975] and Waldinger [Waldinger, 1977] both did further work on the simultaneous sub-goal problem, the most studied bug in HACKER's repertoire. However, none of these systems would meet the criterion of being experts at programming.

This thesis will investigate the expertise needed to represent and understand AI programs. It will also present a reasoning system which knows what it is doing at each step and which uses this information in deciding what to do next. This seems to me to be a basic step toward systems which are capable of modifying and changing their own procedures. [Doyle, 1978b] proposes to investigate such architecture thoroughly. My work should be seen as attacking some of the preliminary technical matters of the investigation; it is not a solution, but merely a stepping stone along the route to self-conscious systems.

What is needed? Here is an example of an episode in which I described - to myself (and my tape recorder) - a plan for developing a certain system:

"This program is going to be doing a network type search. So the main feature of the implementation will be a Conniver-like data-base which in this case doesn't need contexts. There will be a demon feature where we will allow arbitrary number of patterns in each demon. An assertion will be a nested list structure; most of them will be simple. They'll describe the various kinds of relations.

The data-base will take an assertion and regard it as a treelike structure, moving through the structure with the standard tree traversal until it reaches each terminal node. Each terminal will be indexed by a combination of its unique identifier (the MacLisp function Maknum) and its position in the list structure. Indexing means calculating a bucket to put the assertion into. Position is calculated by bit patterns which I'll describe later. These two numbers are combined to form one number and this is used to calculate an index into the array. You do this for each terminal node and insert the assertion in each bucket you get. The bucket will be in increasing order, so the insert will be an ordered-splice-in."

What expertise and knowledge are involved in understanding this description? Certainly one must know something about hash-tables, arrays, list structure, recursion etc. But what about these structures is it important to know? One has to reason about the behavior of such objects. Evidently, I do this by making references to standard kinds of procedures like ORDERED-SPLICE-IN and TREE-TRAVERSAL. What are these cliches? They seem to be more than just particular patterns of code, they are talked about as if they have a more abstract quality. How can we represent these "abstract procedures"? What are the rules for their combination? Certainly any system which hopes to understand complex procedures must have answers to some of these questions.

These tasks seem hard and complex and one might well wonder whether this research will produce a practical payoff somewhere in the not too distant future. I believe it will. Program understanding is crucial not only to AI, but to Computer Science and the computer industry as well. The rapidly growing power of computational hardware, has led to new demands for qualitatively more complex software. Commercial software production is reaching a "complexity barrier" [Winograd, 1973]. I believe that this barrier can only be escaped by using the computer as an intelligent and sophisticated support system for the expert programmer.

4 The Importance of Program Understanding

Section 1.1: The Problem of Program Maintenance

Notes of A Beleaguered Systems Programmer

Software and "software maintenance", has become the major expense of computation. As machines grew larger, faster, and cheaper, the programs which run on those machines have grown more ambitious and complex. But the programmer's tools for maintaining software have not kept pace with this growth.

Why has "maintenance" become so important? The word is probably a misnomer which covers up the real issue: the *evolutionary nature* of the programmer - user relationship. Specifications for large systems are frequently incomplete and unclear; the user doesn't know exactly what he wants. Given fuzzy criteria the designer does the best he can, guessing here, making temporary choices there.

Once a program reaches the stage of initial implementation new desiderata are almost always discovered: "This report should have these 3 extra fields; that one provides extraneous information." New hardware becomes available resulting in changes in the requirements and new opportunities for improvements. In addition, the currently available features suggest new ones which could be implemented if only certain modifications were made.

So while the first implementation is running, work is started on adding features and reworking the last implementation. Running experience reveals the existence of some new bugs which force additional redesign. In this process the programmer again and again finds himself trying to remember whether it is safe to "smash the record" before it is stored, whether any module is using the second bit of the dispatch queue entry, etc. In general he is forced to consider all possible places which might be affected by any proposed change. Of course, one does what one can and *version two* eventually appears.

At this point, the user and the programmer notice that there are new features, a brand new terminal which would allow real time interaction, and of course the inevitable bugs. So while version two is being run, the design evolves again; version three is laid out on the drawing board. And so on...

If the production of software is not to be halted by these problems, new help must be found. Computer science currently has two types of solutions; we shall propose a third. The first type of solution, disciplined programming, consists of improvements which avoid an automation of intelligent human skills. These include languages such as CLU [Liskov, et. al, 1977], ALPHARD [Wulf, et. al, 1976], etc. which attempt to reflect the programmer's intent in the code and to modularize the system so that dependencies are localized. In this kind of programming methodology, errors are minimized and some modifications of the software become simpler. Other efforts short of automation of human skills include the editors, debugging tools, and the like which systems like INTERLISP [Teitelman, 1975,77] and The Programmer's Workbench [Dolotta, 1976] have packaged into integrated language support facilities.

At the other extreme is the proposal to automate the programming process itself, removing the programmer from all but the most high-level design decisions. *Automatic programming* [Balzer, 1973], however, assumes a highly intelligent computer program which is skilled in the problem domain, algorithmic analysis, data structure selection, etc. Some success has been achieved [Green, 1977], [Barstow, 1977], [Manna and Waldinger, 1977]. But a realistic appraisal would suggest that automatic programming systems will not be practically successful without the development of very advanced AI techniques.

I would guess that no truly proficient automatic programming system can exist which is not capable of introspection and self-modification, i.e. of skill acquisition, improvement and development. Once a deep theory of such skills is developed, it might become possible to build automatic programmers which, given advice from an expert human programmer, will improve their skills with practice.

I will present here yet a third approach, called the *Programmer's Apprentice* [Rich and Shrobe, 1976], [Smith and Hewitt, 1975] which is intended to serve as an intelligent assistant to the expert programmer during a program's evolution. The apprentice has only limited skills: it is not yet expert in areas of program design or efficiency, but it does contain a large body of knowledge about programs and fairly sophisticated reasoning capabilities.

6 The Importance of Program Understanding

To use the apprentice one should not be required to provide such a large degree of details that the system would lose all practical utility. Instead I imagine the programmer providing the apprentice with approximately the same volume and type of information that is now supplied as background documentation and in-line commentary. Given this information the apprentice should be able to:

- (1) Modularize the code into appropriate segments each of which has logical coherence and an easily described behavior.
- (2) Derive an explanation of how the behavior of the segments interact to achieve the desired goals of the whole program.
- (3) Deduce which features of the program are crucial and which are gratuitous.
- (4) Relate this program to commonly used techniques of programming.
- (5) Help the programmer decide whether a proposed method could, in fact, achieve the desired goals and whether its sub-units can be fitted together in a coherent manner.
- (5) Detect coding errors as failures of the written code to correspond to the design.
- (6) Index this information for ease of use in program explanation. The apprentice should be able to explain a program in high-level human-like terms.
- (7) Reason about the effect of proposed modifications to the code without having to analyze the entire program starting from scratch.

Our apprentice system is set apart from verification systems like those of [Deutsch, 1973], [King, 1969], [Igarashi, et al., 1975] by its central focus on the evolutionary character of the program design process. As I will explain later, this has led me away from a concern for "proving programs correct". Instead, I have been more interested in building a system which will interact with a programmer during the period of design evolution and which can converse with the programmer in terms which an experienced software engineer would find natural and familiar. The apprentice's goal is to interact with the programmer to develop reasonable designs which meet the engineer's criterion of "good enough" (as opposed the mathematicians criterion of provably correct). The apprentice should be able to analyze program designs at varying degrees of detail. During an initial interactive session it should be able to analyze the program sufficiently to catch obvious bugs. The information obtained by this analysis should be saved so that the apprentice can help assess the effects of changes which the programmer might want to make in the future. Finally, when the cost and time is merited the apprentice should be able to conduct a more thorough analysis and to verify properties of the program.

Section 1.2: An Imagined Scenario

An Idealized Example of Using an Apprentice

A typical interaction between the apprentice and a programmer building an associative retrieval system would look something like the following (Note: as usual the use of English dialogue is a convenient fiction whose only purpose is to make the presentation more comprehensible. Natural language understanding and generation are beyond the scope of this work.) A similar scenario was presented as a "wish list" in [Floyd, 1971]; at the end of the scenario I will indicate how much of my wish list is met by the current system and its foreseeable development.

Programmer: I want to make an associative retrieval system which will be something like the one in CONNIVER. It will store each assertion in each bucket hashed to by one of its leaf nodes. I'm going to start by coding the insert routine for this database. Here's the code:

```
(defun insert (assertion table size)
  (do ((assert assertion (cdr assert))
      (index 1 (plus 1 (times index 2)))
      (car-assert nil))
      ((Null assert))
      (setq car-assert (car assert))
      (cond
        ((atom car-assert)
         (bucket-insert array (hash car-assert index size) assertion))
        (t (insert car-assert table size)))))
```

Note: The code above is written in MACLISP. The `do` function used above is an iteration construct. The first *s-expression* following the `do` is a set of clauses controlling the iteration. Each clause begins with a variable name which is bound locally by the `do`. If there is an *s-expression* following the variable name then the variable is initialized to have the value of this expression; otherwise the variable is initialized to `nil`. If there is a third *s-expression* in the clause, then each time the loop is repeated the variable is assigned the value of this expression. If there are several loop variables, the assignments of their values is done in parallel.

8 The Importance of Program Understanding

The second s-expression following the `do` is a list of s-expressions. The first of these is a termination test which is evaluated at the beginning of each loop repetition. If this s-expression evaluates to a `non-nil` value, execution of the loop terminates. The following s-expressions in the termination clause are evaluated; the value of the last s-expression is the value of the whole `do` expression.

All succeeding s-expressions are the body of the loop and are executed each time the loop is repeated. A `return` expression in the body of the loop terminates loop execution. `return` takes one argument. The value of this expression is the value of the whole loop.

Apprentice: I'm not sure I understand this code completely. It appears that you're doing a *tree-traversal* through the assertion and *filtering* for the terminal nodes which are the atoms. But I don't understand what the purpose of `index` is, nor do I understand why `hash` takes `index` as an argument. The `hash` I know about takes a single object and an array-range and computes an index of that array.

Programmer: In this system `hash` takes two arguments: (1) An atom which appears in the assertion; (2) A number, called `index`, which indicates the position of the atom in the assertion, given that the assertion is viewed as a tree. If the same atom appears twice in the assertion then it will have more than one index; this allows us to distinguish the two occurrences.

Apprentice: I still don't understand. Explain the numbering scheme to me please.

Programmer: The idea is that every node, except for the root, is the immediate child of some other node. Number the root node 1 and then each child is numbered as follows: If the node is the left child of its parent, its number is twice its parent's; if it is the right child of its parent its number is twice its parent's plus 1.

Apprentice: The way you wrote the code when you make the recursive call for the `car` (i.e. the left child when the assertion is viewed as a tree) you don't pass the index of the current node as an argument, so it starts counting from 1 again.

Programmer: You're right. I'll have to change that; that means that the routine will have to take the index as an argument, which means that there should be an initialization function to call from the outside. I'll recode it:


```

(defun insert-1 (assertion table size index)
  (do ((assert assertion (cdr assert))
      (index index (plus 1 (times index 2)))
      (car-assert nil))
      ((Null assert))
      (setq car-assert (car assert))
      (cond
        ((atom car-assert)
         (bucket-insert array (hash car-assert index size)
                        assertion))
        (t (insert-1 car-assert table size (times index 2))))))

(defun insert (assertion table size)
  (insert-1 assertion table size 1))

```

Apprentice: One further question. I assume, given its name, that `BUCKET-INSERT` inserts its third argument into the bucket indexed by its second argument. You're inserting `ASSERTION` on each call, which is not the same object all the time. My understanding of this type of hash-table is that an assertion should be in each bucket that its terminals hash to. But when you make the recursive call for the left nodes, `ASSERTION` becomes the current sub-tree of the assertion.

Programmer: You're right again. I should make the whole assertion an argument to the procedure as well. Can you make that change?

Apprentice: Yes, consider it done. How about the `BUCKET-INSERT` routine; would you like to design that now?

Programmer: OK. In this design, buckets will be ordered lists of assertions. `BUCKET-INSERT` will fetch the bucket indexed by its second argument and then do a `LIST-INSERT`. This will be the standard *trailing-pointer-enumeration* which tests each item to see if its `UNIQUE-IDENTIFIER` is larger than that of the assertion. I'll use LISP's `MAXNUM` to implement the unique identifier. When the first larger member of the list is found, I'll splice in the new assertion.

Apprentice: I understand the design, but there is a mistake. If the new assertion is smaller than any element, then it can't be spliced in.

Programmer: In that case, cons it onto the front of the list.

10 The Importance of Program Understanding

Apprentice: That won't work either, since you want the assertion to be a member of the bucket indexed by the hash you've calculated. But if you just cons this new assertion onto the front of the list it won't be in the list pointed to by the table. The item of the table in that position will still be the old list which is the cdr of the one which contains the new item.

Programmer: OK. If this case is reached we'll not only cons the element on, but we'll also store this new list in the appropriate position of the table.

Apprentice: The design now is correct.

Programmer: Here's the code:

```
(defun bucket-insert (array index item)
  (do ((current (arraycall array index)(cdr current))
      (previous nil current)
      (unique-id (maknum item)))
      ((null current)
       (cond ((null previous)
              (array-store array index (list item)))
             (t (rplacd previous (list item)))))
      (cond
       ((> (maknum (car current)) unique-id)
        (cond ((null previous)
               (arraystore array index (cons item current)))
              (t (rplacd previous (cons item current)))))
       (return t))))
```

Programmer: Now I'd like to do the LOOKUP. I just realized that LOOKUP is very similar to INSERT. It also does a tree-traversal through the list structure, calculating indices and fetching buckets. I think that this ought to be modularized so that there will only be one place where this indexing is done. I guess what I'd like to do is write a subroutine called INDEX which takes an assertion, the table and the size and the returns the list of buckets which this assertion indexes to.

Apprentice: The current INSERT routine is a tree-traversal which produces a sequence of indices which are handed to BUCKET-INSERT. Each occurrence of BUCKET-INSERT fetches the bucket. You can re-arrange things so that INDEX will do the TREE-TRAVERSAL, producing the sequence of indices; each of these is fed to BUCKET-FETCH to get the bucket, and each of these can then be fed to a *list-accumulator* to produce a list of

buckets. Here's the code:

```
(defun index-1 (assertion table size index bucket-list)
  (do ((assert assertion (cdr assert))
      (index index (plus 1 (times index 2)))
      (car-assert nil))
    ((Null assert) bucket-list)
    (setq car-assert (car assert))
    (cond
     ((atom car-assert)
      (setq bucket-list
              (cons (arraycall array
                               (hash car-assert index size))
                    bucket-list)))
     (t (setq bucket-list
               (index-1 car-assert table size
                       (times index 2) bucket-list))))))

(defun index (assertion table size)
  (index-1 assertion table size 1 nil))
```

Programmer: Can you fix INSERT to call INDEX, rather than doing the indexing itself?

Apprentice: It would seem so. Since INDEX produces a list of all the correct buckets, if I do a standard list-enumeration of these I'll produce a sequence of buckets to be handed one at a time to BUCKET-INSERT. So the general plan is still the same. But, BUCKET-INSERT will have to be changed since it now expects its input to be an array index not a bucket.

Programmer: Change it to make it expect the bucket.

Apprentice: Now you've got problems. In the special case where the item being inserted is smaller than any presently in the bucket, we had to store the new bucket back into the table. But BUCKET-INSERT in this new version doesn't have the index of the bucket.

Programmer: Oh well! Actually, I wanted to change the representation of the buckets a little anyhow. I want the first item of the bucket to be a count field and then the list of members to be the rest of the list. I think if you'll check, it will turn out that this removes the problem since the item to be inserted will always have to come after

For Complex Program Understanding

12 The Importance of Program Understanding

the count field; you can always do this by side-effect. The LIST-INSERT routine will have to be initialized differently, starting with the PREVIOUS pointer pointing at the whole bucket including the count field and the CURRENT pointer pointing at the COR of this list which is the part with all the items in it.

Apprentice: Yes, that will work, except that I can't prove that the table will always have such a structure in each slot. Arrays are initialized to nil, not buckets as you just described them.

Programmer: I will write an initialization routine later which will set up the table to have empty buckets in each slot.

Apprentice: What should be in the count field of the bucket? There is no code to maintain it yet.

... and so on

[Note: this dialogue was extracted from a transcript of several coding sessions].

This scenario clearly is more ambitious than anything currently implemented. In particular, the language and discourse expertise implicit in this scenario are not even part of my current research goals. However, the basic facilities in this system are now under development. The apprentice system I have shown consists of four main facilities. First, the programmer proposes designs which the apprentice checks for logical consistency. In chapter 7 I show an example of such an analysis which was conducted by the first implementation of REASON. In the case where a design is incorrect, however, the scenario shows REASON framing high-level descriptions of the problem. This facility is not yet developed.

The second facility shown in the scenario is the ability to determine whether the actual code corresponds to the design. This facility is not part of REASON at all; in [Rich & Shrobe, 1976] we described an initial design of a recognition system which could conduct such an analysis.

The third facility shown in the scenario is the use of pre-proven fragments to analyze a program's behavior. This is coupled with the use of *temporal collections* to segment the system into these fragments. Chapters 8 and 9 present a detailed formalism for this kind of analysis. [Waters, 1978] reports on an implemented system which conducts such an analysis for mathematical FORTRAN programs. The use of

plan fragments to guide the logical analysis is not yet implemented in my system.

The final facility is shown in reasoning about program modifications. This is discussed in chapter 13. My current system is only part of the way to being able to handle such reasoning. As the system makes deductions it records the logical dependencies used in each step. Furthermore, once it has conducted a complete proof it analyzes these into a summarized form called *purpose links*. These, together with the complete proof are the basis for further reasoning about modification. However, the techniques for using these representations are still being developed.

Throughout the scenario, the apprentice exhibits considerable expertise about data structures and side effects. In Chapters 11 and 12 I discuss the techniques used to conduct such an analysis. The basic framework shown in these chapters is now well worked out, but is still in the process of implementation. The earlier version of REASON could conduct similar side effect analysis, but was far less robust and flexible.

In general, the reader should remember that the above scenario is a wish list and that the remainder of this thesis is a progress report on the research required to implement the facilities of the wish list.

Section 1.3: The Research Content of This Thesis

The apprentice represents a practical, medium term research goal in which many of the issues of program understanding can be explored. The representation of programs, the ability to understand the underlying logic of a program and to reason about the effect of program modifications are crucial prerequisites to the development of a self-conscious system capable of serious skill acquisition. I will develop in this thesis a representation called *plans* for the logical structure of programs and a reasoning system which follows a discipline of explicit representation of its control strategies. This will allow the system to examine its own control state and to choose what to do next based on that examination.

Throughout the scenario above we saw the apprentice and the programmer referring to a shared body of knowledge about standard program structure. The apprentice talked about *tree-traversal*, *list-accumulation*, *filtering* certain elements. The middle section of this document will develop the *plan* formalism for representing such notions and will present examples of some useful standard plans. Influenced by the work in [Waters, 1977] this formalism has been extended from that in [Rich and Shrobe, 1976] to allow plan fragments which produce and consume sequences of objects distributed in time. The importance of this notion can be seen in the scenario above where the apprentice develops the code for INDEX from that for INSERT. Having done this the apprentice notices that INDEX produces a list of buckets which is the same sequence of values as that produced by the internal TREE-TRAVERSAL and BUCKET-FETCH fragments which were internal to INDEX. When grouped appropriately, a call to INDEX followed by a standard LIST-ENUMERATION is identical to the internal fragments of INDEX. In the last section of the thesis I will consider how to analyze the effects of such program modifications and how to maintain consistency as procedures evolve through the design process. I will show how the analysis of programs into plan fragments can greatly reduce the complexity of understanding modifications.

The programs which I present in this thesis involve side-effects on complex and shared structures. Analyzing this kind of program is a very tedious process which people simplify using many heuristics. In chapters 11 and 12 I will show how the reasoning system can analyze side-effects at varying levels of detail which correspond to the levels which people seem to use. This will allow the system to develop an understanding of what the program is intended to do, before it is forced to determine what it actually does in all the possible "screw-ball" cases. The system is capable of going back and being more careful in its analysis, using the information from the first

analysis to guide the second. The use of dependency information and non-monotonic logic [Doyle, 1978] to conduct side-effect analysis is unique to this thesis.

The work reported on here is part of an ongoing project to develop a working Programmer's Apprentice. Charles Rich and I began in this work in 1974; many of the ideas presented here were developed jointly. It is hard to identify all those places in this thesis which have been influenced by Rich's [Rich, 1977] ideas. The Programmer's Apprentice project was later joined by Richard Waters who used many of our initial ideas to analyze numerical FORTRAN programs [Waters, 1976]. Waters found it very convenient to think of loops as being built up by a series of Plan Building Methods. In his view a loop consists of a *nucleus* which produces a sequence of values. Embedded in this nucleus are various *augmentations* which consume the sequence of values produced by the nucleus. Augmentations can be made to operate on a restricted sequence by including a *filter* to eliminate certain elements from consideration.

Waters' ideas find their way into this thesis as a more general notion of *temporal-collections* which may serve as the outputs and inputs of segments. Any recursive program can be viewed in at least two ways, involving two distinct segmentations: One of these called the *temporal-viewpoint* involves a cascade of segments passing such temporal collections; the other, called the *surface-viewpoint*, is simply an aggregation of the code into modules. Some features of the program are made clear by the temporal viewpoint while others are seen more easily in the surface viewpoint.

The program REASON reported on in this thesis has gone through several incarnations. As part of my earlier work with Rich, an initial version of REASON was designed and reported on in our Master's Thesis [Rich and Shrobe, 1976]. That version was completely coded and worked as reported. However, the earlier version was rather cumbersome and ill-suited to the recognition tasks for which it was intended. During the later part of the development period of the first version of REASON [Stallman and Sussman, 1976] introduced the use of dependency networks in their electronic circuit analysis program EL. The dependency network was then extended and built into a separate package called the Truth Maintenance System in [Doyle, 1978]. Although the first version of REASON maintained dependencies, it did not have a truth maintenance system which used these to any advantage. Doyle and others, however, built a simple problem solving language, called AMORD [DeKleer, et. al., 1977] which did interact with the TMS.

For Complex Program Understanding

16 The Importance of Program Understanding

The current version of REASON is being developed as a program written in a variant of AMORD. I have had enough experience with the first implementation and the partially completed new version to report on this work with confidence.

In the chapters 14 and 15, I will attempt to evaluate this admittedly partial work and to compare it to other, more developed systems for program understanding. I hope that the reader will find the tedium of working through this document rewarded by at least an occasional glimpse of something promising.

Chapter 2: An Engineering Theory of Evolutionary Design

The scenario of the last chapter emphasizes the evolutionary character of the design process. I believe that the key to supporting such an evolutionary interaction is the development of powerful techniques for *program analysis*. Analysis is the process of decomposing a program into coherent modules such that the behavior of the whole artifact can be understood in terms of the behavior of its parts.

The dominant form of analysis in current computer science research is the Floyd-Hoare verification techniques, especially as developed by [Igarashi, et. al., 1973]. In this technique the basic unit of decomposition is the programming language primitive whose behavior is specified by axioms written in Hoare's logic. Each such axiom provides a method for transforming a post-condition of a programming language primitive into a pre-condition (or vice-versa). Verification usually proceeds by stating a pre-condition and a post-condition for an entire program. These are combined using a process known as Verification Condition Generation (VCG). VCG passes the post-condition back over each language primitive of the program in turn; the statement arrived at when the modified predicate is finally passed over the first primitive of the program has the property that it must be true on program entrance if the original post-condition is to be true on program exit. Finally, an implication is formed from the pre-condition of the whole program and this new statement. If this implication can be proven, then the program must exhibit the behavior specified by the pre- and post-conditions.

Given the attention paid to verification techniques in recent years, one might think that they are sufficiently powerful to help manage the problems of evolutionary design. I feel that such a conclusion is unwarranted. Program proving techniques will play an important, but limited role in supporting incremental design and evolutionary programming. It is my feeling that the techniques now in existence have been designed with a particular kind of program - namely algorithms - in mind and that there are a large number of distinctions between such programs and the software systems which I am interested in. This difference of concern has lead to a difference in perspective and methodology which underlies this entire document.

1. Much of this material was originally written as part of a proposal to the National Science Foundation. I acknowledge and appreciate the extensive editing by Charles Rich and Richard Waters which went into those sections.

For Complex Program Understanding

Section 2.1: Type of Programs -- Algorithms vs. Systems

We can identify two different kinds of programs: algorithms and systems which differ along a number of dimensions. Each kind of program is valid and important to computer science, but they present different demands and requirements. I believe that program proving is most useful and necessary for the domain of algorithms while the techniques I introduce in this document are more useful for the analysis of systems.

Typically, an algorithm is a relatively short program which can be precisely and concisely specified. Specifications for an algorithms often are much shorter than the program text. For example, the Euclidean GCD algorithm occupies about 7 lines of code in any recursive language; its specification is of about the same length: the answer divides both inputs and it also is divisible by any other common divisor of the inputs. The Knuth-Morris-Pratt or the Boyer-Moore string matching algorithms occupy roughly 100 lines of code but have a very short specification: the answer returned is the position of the first string in the text which matches the input pattern.

A second feature characterizing algorithms is that they exhibit a clever underlying logic which *requires* proof. The intricacies of either of the string matchers mentioned above would lead one to doubt whether they worked unless a rigorous proof were presented. Indeed, the cleverness underlying any particular algorithm makes its code quite different than of other algorithms. Thus, one finds few familiar cliché's in the code. Instead one must work hard to find an explanation for the function of each line. In addition, since algorithms are meant to be used as components of other programs, it is crucial that they be known to be correct; a single mistake could have thousands of repercussions.

Algorithms are built to satisfy a precisely stated specification which has general utility. The specification is not subject to change or reinterpretation. An algorithm is not an evolutionary program. Euclid's and Pingala's algorithms have survived in essentially unchanged form for more than a millennium.

A final point about an algorithm is that it frequently represents an extremely optimized method for achieving a very common task. This optimization is achieved through clever and often obscure techniques. But in the case of an algorithm this is allowable and even desirable. The algorithm is published with an explanation; it is not intended to be modified and therefore intricacy is appropriate if it leads to an improvement over previous techniques.

One might be inclined to think of a system as nothing more than a large collection of algorithms. However, the description above should make it clear that the whole is more than just the sum of its parts. Each of the characterizing aspects of algorithms are in fact untrue of systems. Software systems are large programs with specifications and other related documentation exceeding the size of the code by an order of magnitude. The specifications are not crisp, well defined, or permanent. Indeed, they often are tied to social and institutional practices which change for reasons having nothing to do with computation. Financial and management systems are dependent on tax codes, business practices, etc. Military related systems depend on the arms technology and defense strategies of the world powers. When specifying systems which function within such a nexus, it is impossible to state precisely what is to be done; one instead states some criteria which must be met and others which are suggestive of less crucial but desired behavior. In any event these criteria change, and the system is forced to evolve to meet the new criteria.

Although much can be done with methodologies for requirements definition, experience in the military and in industry suggests that the effect of such methodology has a significant but limited impact. It is an empirically undeniable fact that specifications are never completely elaborated and that they evolve with the life of a system. Even in the most careful of system designs the product passes through many iterations of design and coding before something acceptable is developed. In the next section I will present an argument for why this must be the case.

Systems also differ from algorithms in the degree to which they involve clever and intricate logic. Typically a system of programs is made up of a large number of relatively small modules, each of which involves routine and mundane code. There is a vocabulary of cliché's out of which such code is built and the experienced programmer can analyze such routine coding patterns by inspection. Occasionally something idiosyncratic is thrown in but even these are usually simple to understand. Even at higher levels of the system, sub-modules are combined in routine ways. Verification of such modules would be conducted mainly as a method of isolating coding mistakes such as fencepost errors and typos.

The complexity of a system does not primarily arise from the use of locally intricate strategies, but rather from the sheer number of interactions between modules. These make it difficult to assess the effect of a proposed change to the system since each module may enter into purposeful relationships with many others. Systems tend to reach a point where the volume of these interactions overwhelm unaided human

For Complex Program Understanding

abilities to manage the complexity. Once this point is reached changes to the system produce more harm than good. Rather than continuing to evolve, the system is frozen and a new system is commissioned.

In summary, we may distinguish between algorithms and systems along two major dimensions. Algorithms are permanent, almost mathematical objects, which are not subject to frequent modification of either code or specification. Systems are impermanent, evolutionary programs of little mathematical interest. The complexity of an algorithm is largely due to the use of locally intricate and clever strategies; the complexity of a system is due primarily to the sheer volume and number of interactions between modules.

These distinctions lead one to see the need for different kinds of automated design tools. The designer of algorithms needs the use of proof checkers, theorem provers and verification systems. While these serve a useful role for the systems designer as well, they are not his bread and butter. Instead he needs tools to help him modify current designs to meet incrementally new requirements. Given the size of a software system, one cannot tolerate the delay and expense of a completely new analysis every time such an evolutionary modification is desired. One instead requires incremental processing in which a small change in the design should require only a small amount of reprocessing to achieve an adequate analysis.

Even if a program can be proven correct, and even if this can be done in an incremental fashion, there is still a problem. The sheer volume of information developed during the proof of a software system renders the information useless unless the analysis structures the information so that it is comprehensible to a human. When a system designer sets out to make a modification, only a tiny fraction of the information is actually relevant; the system designer needs tools which can produce just this information and no more. My work is directed towards the production of such tools.

Section 2.2: What Characterizes Evolutionary Change?

As we saw in the scenario of the last chapter, the user repeatedly proposes designs and then debugs these designs until he is convinced that they achieve the desired goal. In some cases he even goes so far as to reorganize the program, breaking down some module boundaries and erecting new ones in their stead. For

example, the code for indexing an assertion is extracted from the `INSERT` code, modified and then made into a module which is called from the `LOOKUP` routine. At another point the user decides to change the structure of the `BUCKETS`.

In each of these cases the proposed modifications have ramifications which reach beyond the boundaries of any modules apparent in the code, yet the programmers clearly think of them as incremental or evolutionary changes. I wish to contrast these evolutionary changes with the situation in which the programmer cannot accommodate whatever change he is considering within his current conceptualization of the system and so redesigns "from scratch". In an evolutionary change, the main parameters of the current design are left as they were; only some small set of changes is needed to achieve the desired goal. In redesign, the entire structure of a program is blocked out anew.

How can we tell which of these categories a change will fall in? This is a key question since I want my system to be prepared to handle those changes which an experienced programmer will think of as evolutionary. The received wisdom in programming methodology is the principle of the modularization. This principle (which is the computer science version of "a place for everything and everything in its place") suggests that if module boundaries are carefully arranged so as to localize each design choice to a single module then all evolutionary changes can be handled by local changes to a small number of modules. However, there are clearly evolutionary changes which do not fit into this paradigm. As I have already mentioned, the scenario contains examples of the programmer making modifications in which the module boundaries are rearranged, yet these changes are clearly thought of as evolutionary.

The principle of modularization suggests that evolutionary changes are those which are local to a module. Although, I believe that this notion is overly rigid, I do believe that the notion of locality within a decomposition is the crucial idea which characterizes those changes which can be treated as evolutionary modifications. In the next several sections I will develop the following thesis: Engineering analysis consists of the use of partially accurate models to allow a system to be decomposed into multiple, overlapping, tangled hierarchies. A modification will be perceived as evolutionary if there is at least one decomposition such that within its segmentation structure the effect of the modification appears local.

Section 2.3: Why Is Evolutionary Design Necessary?

Why don't we just design correct programs to begin with and dispense with the expense of design iterations, debugging, and evolution? There are those, for example [Dijkstra, 1976] who believe that such a fault-free methodology is both desirable and possible. In this view, one would start with specifications for a program's behavior and refine these in a top-down step-wise manner until a correct program had been reached. It is claimed that by carefully stating the invariants (for example on loops) before they are coded, one can assure a high degree of reliability for the code produced. This may be summarized by saying that one should have a proof of correctness in mind when one begins to code.

As far as this goes it presents little to argue with; however the methodology provides little concrete guidance as to how one should develop the design and proof of correctness to begin with. I suspect that if this approach is suitable at all, it is only useful for the creation and implementation of algorithms. As I have observed, the process of algorithm development is quite different than that used for the design of large software systems. Algorithms are the result of months (or years) of research; when a researcher has the insight for a new algorithm, he can then proceed through a top down design process in which his insight is elaborated into a design for the coding. This can and should include specifications for the sub-modules of the algorithm. Such careful specification and elaboration of the algorithm's design can then lead to a correct or nearly correct coding of the program.

The design and construction of large software systems is quite different. I have already observed that systems are evolutionary by nature. One reason for this is external; the design of a system often depends on social and institutional practices which change quite frequently. However, there is also an internal, cognitive reason why systems are designed incrementally, namely that the cognitive complexity of the task allows no other approach. The designing of a software system is, in my view, a form of problem solving not very different from that used in a conventional engineering disciplines such as electrical engineering or even in common sense reasoning. The overriding goal of such forms of reasoning is to manage and reduce the complexity of the design task to the point where human cognitive powers are adequate to produce a reasonable solution. Much of Artificial Intelligence research on problem solving has consisted of the development of paradigms which account for this type of reasoning. These center around the related ideas of decomposition, modeling, and debugging as intrinsic parts of the planning process.

Common sense reasoning and engineering problem solving share a need to limit the complexity of the planning space. In both these domains if all possibly relevant details were to be considered at once they would overwhelm human cognitive capacity. Thus, rather than trying to guarantee a perfect answer from the start, one works for an answer which is close enough and then modifies this to fit the actual needs. If one does not take this approach but rather insists on perfection from the start, the planning process would stall out at the first step.

The goal of a problem solver is to piece together a collection of actions which will achieve a specified set of goals. Typically, the problem solver only has to achieve these goals given that certain conditions hold in the initial world state. This collection of actions is called a "plan" and consists of several forms of information: First, a set of sub-steps and their behavioral descriptions; Second, a set of constraints on the ordering of sub-step execution; Third a means of propagating information between the sub-steps. Finally, and most importantly, the plan includes an explanation of how the segments interact to achieve the desired goals. Notice, however, that the sub-steps are not necessarily primitive actions; the problem solver may have to attempt their solution recursively.

The earliest planning systems used the paradigm of heuristic search in which the problem solver repeatedly tries to take a single step from its current world state to another state which is hopefully closer to the goal. This approach was used in systems like GPS [Newell & et. al, 1959] and STRIPS [Fikes & Nilsson, 1971]; however, it was found to lack sufficient power for task of even moderate complexity. Minsky's suggestion of the notion of "islands" [Minsky, 1961], however, led to a new paradigm which was partially embedded in the PLANNER language [Hewitt, 1972].

The key insight in PLANNER is that a reasonably knowledgeable problem solver will often recognize the form of the answer; having done so it can propose a partially instantiated plan immediately. Such plans are used as the starting point of the solution with the problem solver recursively attempting to synthesize a plan for the sub-steps by recognize the form of their answer. This method of problem solving (called *Planning by Recognition of The Form of The Answer*) was formalized in the Planner programming language and was improved upon in its descendants Conniver and QA4.

Within particular domains of expertise the paradigm of Planning by Recognition of the form of the answer is facilitated by the development of an engineering vocabulary which conveniently captures the abstract form of most problems. In particular, within programming domains one often can identify an "intermediate vocabulary" of programming abstractions which constitute the building blocks out of which a large percentage of the known techniques of particular domains are built.

When viewed from the perspective of analysis, the Recognition Paradigm takes the form of Analysis by Inspection. As a planning paradigm Recognition decomposes the problem into a pattern of sub-steps by recognizing the form of the problem. As an analytic technique, Inspection reconstructs the form of the problem by recognizing the pattern of sub-steps in the device. Both of these rely on the existence of a powerful body of "standard plans" which reflect the common ways of achieving those goals whose form is understood. The existence of this body of knowledge reduces the heavy cognitive cost of heuristic search to the much less burdensome price of searching a "plan library".

However, even the paradigm of Planning by Recognition does not adequately model human problem solving behavior on very complex tasks. Yet another paradigm, that of *Planning In an Abstraction Space* [Sacerdoti, 1973] must be added. In this paradigm we add to the above notions a further idea, that of modeling. An abstraction space is a model of the real world in which some important details are intentionally (or otherwise) omitted. Planning is first attempted in such an Abstraction Space. If a completely developed plan is formed in the abstraction space, then the process advances to an attempt to modify the plan to function in a less abstract space.

Notice that this recursion of planning and refining in a hierarchy of abstraction spaces is a different recursion than the recursive invocation of the problem solver on the subgoals. In the later recursion the metric is the size of the task, in the former it is the accuracy of the modeling space. An important consequence of this paradigm is that as one proceeds through increasingly accurate models, a new plan is formed by incremental modification of the plan produced by the preceding stage. One implementation of this paradigm was embodied in the Abstrips program [Sacerdoti, 1973], a descendant of Strips. Comparisons between the two programs showed that Abstrips could outperform Strips by a factor of 4; as the problems grew harder the difference between the two systems became even more pronounced.

Abstrips, however, had only a very weak method of modeling the real world; its only abstraction consisted of weakening the preconditions of its built-in operations. Thus, its only debugging technique consisted of splicing set-up steps into the abstract plan.

Sussman's Hacker also identified a second reason for the indispensability of debugging. Suppose that a problem solver is presented with a goal for which it has no plan in its library. In this case, the problem solver should attempt to reformulate the problem statement so that it can be decomposed into parts whose solutions can be found by Recognition. However, when this decomposition is made there is always the possibility of destructive interference between the plans for the various sub-parts. Furthermore, until one has found plans for each sub-goal separately, one cannot tell whether they interact. Inherently one is faced with the need to debug the total solution to remove destructive interference between the sub-plans.

I believe that these paradigms explain the mechanisms used by people to manage the complexity of planning in large and complex domains. One first constructs a mental model of the domain in which many details have been omitted. This produces a search space of considerably smaller size in which it is computationally feasible to derive a plan. This plan, like every other, has a "proof of correctness" (or an explanation of how it achieves its goals); however, this "proof of correctness" might actually be incorrect since it depends upon assumptions in the model which may violate facts in the real world. Nevertheless, these fictions in the modeling process are extremely valuable; without them the complexity of the problem would prevent one from building a plan at all. This "almost right" plan is refined by developing a more accurate model of the situation and then using the current "proof of correctness" to guide the debugging process. As the Abstrips program indicated, developing the plan in an abstraction space and then debugging it is a computationally cheaper option than attempting to develop a correct plan directly. It is for this cognitive reason that software must be designed in an incremental, evolutionary manner.

If computer based design aids are to be of assistance to software system designers, they must take cognizance of the nature of the design process which I have outlined. Design aids must satisfy two criteria: First, they must be able to reason about abstract plans and their hierarchical structure. Given any world model the design aid must be able to check whether a proposed design will achieve its goal. Since the plan development process is a recursive one in which the sub-steps of a plan are themselves candidates for plan synthesis, the design aid must be able to understand

a proposed plan even if its sub-steps have not yet been designed. However, the constraints imposed on these sub-steps must be remembered so that they can be checked when the plans for the sub-steps are formulated.

The second major criterion that such a system must meet is its ability to deal with plan editing, modification, and debugging. A plan is initially developed to work under the assumptions of the abstract model; when these assumptions are revised to more closely correspond to the real environment or when the environment itself changes, the logic of the original plan must be examined to see what dependencies are no longer valid. Thus, the design aid must be a dependency based reasoning system capable of sophisticated belief revision processing.

The problem of managing evolutionary design faces engineers in all disciplines. But it is particularly acute in computer science for two reasons. First, computer science is a young field without the maturity and experience of civil, mechanical or electrical engineering. In a sense there is as yet no engineering discipline. Secondly, software engineers deal with a peculiar problem in that the major constraints one deals with are not physical but social. Since social phenomena are more transient than physical laws, the modeling process in software system design is unusually hard and inaccurate. This suggests that software engineering should look to the more mature engineering sciences which have developed sophisticated techniques for managing the complexity of their fields. We will see that the paradigms of problem solving developed in Artificial Intelligence research have their counterparts within these mature engineering domains.

Section 2.4: What Do Engineers Do?

One might think that engineering is mainly concerned with the optimization of numerical parameters within physical systems. If so, computer science would have little to gain from the study of the methodologies used in engineering. Indeed, engineers do conduct such activity, but this is only a small part of what engineering is about. Engineering is mainly concerned with limiting the complexity of analysis. [Bose & Stevens, 1965] give the following account of the engineering exploit:

A physical problem is never analyzed exactly. This is a consequence both of our inability to describe a physical situation completely and of the increasing complexity of the analysis as greater accuracy is demanded. A problem that involves events in the real world is always approached by making simplifying assumptions that hold only approximately, thereby forming a *model* of the events under study. The problem then reduces to that of analyzing the model. If the assumptions by means of which the physical situation was reduced to the model are reasonable, then our analysis should produce results that correspond to observed events, and the same type of analysis should be useful in predicting the behavior for other similar physical situations.

I have identified three areas of technique which seem to be common to all engineering disciplines and which provide fruitful starting points for the development of a similar technology for software engineering. These areas are (i) The construction of "almost accurate" models which reduce the complexity of a pure physical analysis by introducing tolerable inaccuracies; (ii) The decomposition of complex systems into several possibly overlapping almost hierarchical organizations in which aspects of the behavior of the whole artifact may be simply inferred from the behavior of the sub-systems. (iii) The development of a vocabulary of characteristically useful intermediate constructs which allow analysis by inspection.

Engineering models reduce the complexity of an analysis by omitting details not relevant to the task at hand. Electrical engineers, for example, use models of transistors which describe their behavior accurately enough given that the transistor is known to be operating within a certain range of frequencies and power. Such models will, however, produce grossly incorrect result when used outside the range of their applicability. Thus, in analyzing a system more than one model of a particular component may be used, each model explaining the components behavior within some range of operation.

In the domain of programming one also needs to model the behavior of various parts of a system. Richard Waters and I have developed a modeling technique, called temporal abstraction, in which some aspects of a system's behavior are made quite easy to understand. For example recursive programs can be temporally abstracted into a simpler non-recursive programs in which sequences of data are communicated in parallel between sub-segments. I will give an overview of this technique later in this chapter and will present in thoroughly in Chapter 8. In the temporal model of the

program some ordering constraints are omitted. Thus, a second model corresponding more closely to the surface features of a program is also needed.

Engineering modeling makes a trade off between accuracy and ease of analysis. In order to be able to make the analysis the engineer is willing to introduce "tolerable inaccuracies". Engineers don't have to be perfectly correct, only "close enough". However, when a model is used inappropriately conclusions can be reached which exceed the threshold of tolerable errors. One must, therefore, maintain a record of how each conclusion was reached so that a debugging process can be invoked to identify the source of the error and to substitute a more appropriate model. I will present a program reasoning system which uses the Truth Maintenance System [Doyle, 1978] to maintain this information. This allows our system to incrementally reanalyze a program when its original models were found to be too sweeping in their omission of details.

A second area of technique common to many engineering disciplines is the decomposition of larger systems into a (possibly overlapping) hierarchy of sub-systems. Each sub-system is given a simple description which describes only those aspects of its behavior which are relevant to other sub-systems. We may then regard the whole artifact as a loosely coupled network in which the behavior of the whole system may be deduced from the descriptions of each subsystem. Often, however, it is necessary to decompose a system in more than one way in order to derive convenient explanations for all of its behavior. In electrical circuit analysis, for example, one makes one decomposition to facilitate the DC analysis and a second decomposition for the AC analysis. A single component may be present in both decompositions playing different roles depending on which decomposition it is viewed from.

Engineering decomposition techniques include some of the most elegant analytical methods of all science. Norton and Thevenin's equivalence theorems for electrical networks allow one to decompose any electrical network into a collection of two-terminal devices which are accurately modeled by a single source and a single impedance.

Perhaps because decomposition is such a basic strategy, it is also a relatively advanced technique in computer science. The use of sub-routines as procedural abstractions which are described by their input-output behavior is well established. Data abstraction techniques allow a second type of decomposition. Typically, these techniques are embodied in the features of a programming language. While I

recognize the significance of such efforts I also note a drawback. Analysis frequently requires multiple decompositions of a single system; however, a programming language requires that the system be represented by a single decomposition which is most often correlated with the imperative structure of the system.

The third major type of engineering methodology involves techniques to facilitate analysis by inspection. For each design problem an engineer must establish the form of the answer. Frequently the most powerful aspects of an engineering discipline exist to facilitate analysis by inspection. In electrical engineering, for example, the notion of complex impedance allows the inspection techniques which were first developed for resistive circuits to be applied to circuits involving inductances and capacitances as well. Thus, a single set of abstract forms, such as the notion of a voltage divider, can be applied to a much broader class of circuits. Without this technique, the far more complicated methods of differential equations would be required.

Given such techniques it becomes possible to catalogue the various forms of problems and their typical solutions. This is done by developing a craft or engineering discipline with an associated vocabulary of macroscopic constructs. Although there are virtually an infinite number of combinations of the primitive objects of any discipline, most of these are not useful. However, a much smaller set of combinations turn out to have sweeping power within particular domains. These form the "standard plans" of a domain; they are the terms of the engineering vocabulary. Lisp programmers, for example, have a relatively rich vocabulary including ideas like "cdring down a list", "tree traversal", "searching a sequence of values", "consing up an answer", etc. In chapter 9, I will discuss the process of analysis by inspection; chapter 10 will present a brief catalogue of some useful data structures and chapter 13 (in passing) will present a description of some typical procedural plans.

An engineering approach works with such higher level notions since such descriptions reduce the complexity of making sense of a device. The various techniques which have been mentioned so far interact to allow an analysis to decompose the system into components whose behavior conveniently explains the behavior of the whole. For example, the construction of a temporal model allows the system to be decomposed in a manner which separates the process of generating a collection of objects from the process which consumes these objects. Once this decomposition is performed, it frequently becomes trivial to analyze the components by inspection. In this case, we have an interaction between modeling, decomposition and recognition. In chapters 12 and 13 I will show another modeling technique which

similarly reduces the complexity caused by allowing side effects on shared data structures.

By using these techniques, it becomes possible to impose a very rich structure on a program. This structure includes: several decompositions with mappings between them, simplifying modeling assumptions, and recognition mappings which explain how a particular fragment corresponds to a prototype from the plan library. This vast quantity of information is unified by the use of a dependency based reasoning system which records all logical dependencies which it discovers in a special data base. These dependencies may then be consulted at any time to discover the possible ramifications of any proposed modification. My thesis is that the techniques outlined above facilitate an analysis in which any change which a programmer would regard as evolutionary is localized within the module boundaries of at least one decomposition. Once a modification is localized within some decomposition the task of assessing the impact of the change becomes cognitively manageable since the decomposition renders irrelevant all but a small fraction of the information.

In the remainder of this chapter I will present a somewhat more detailed overview of the techniques which I have developed along these lines.

Section 2.5: Plans and Teleology

Whether designing or analyzing a device, an engineer must have a representational system within which it is possible to utilize and coordinate information derived through the techniques described above. In most engineering disciplines there is a notion of the "design plan" which forms a skeleton around which all of this information is arranged. Of all the issues discussed so far, the design plan is the one least well addressed by other current work in computer science.

In traditional engineering or software engineering, the behavior of a device can be described in two ways. Some properties of a device are independent of its context of use. These properties constitute the *intrinsic* description of the device. The LISP function `APPEND` can be described intrinsically by its input-output behavior of returning the concatenation of its arguments. A device may also be described by its role or purpose in the plan for a larger mechanism. This is its *extrinsic* description. `APPEND`, for example, may be used to produce the union of two disjoint sets represented as lists.

A single part may have several extrinsic descriptions corresponding to multiple needs that it satisfies in the larger mechanism. A copying garbage collector such as [Minsky, 1963] uses the same array of space as both the destination for reclaimed cells and as a queue in a breadth first tree traversal of the space of used cells. There may also be several plans for a given device, describing its structure in different dimensions. In this situation, each part has the potential for one or more roles in each plan.

The essence of understanding a mechanism is knowing the purposes of each part. This involves building a description of the mechanism which matches each part with its roles in the appropriate plans. Each role in each plan must be filled by some part of the mechanism and the intrinsic properties of that part must satisfy the extrinsic properties of its roles.

Certain plans or plan fragments can appear as part of the plans for many different devices. For example, the depth first tree traversal plan fragment appears in several of the modules coded in the scenario. However, understanding the teleological structure of a plan fragment (which may be very difficult) need only happen once. Any properties of the plan fragment which can be discovered, are known to hold wherever the plan is used. These common plan fragments serve as an "engineering vocabulary".

Section 2.6: Representing Plans

Supporting a programmer during design evolution requires the apprentice to reason about program designs before they have been committed to code. This requires the apprentice to have a program representation which is independent of the choice of programming language. In our Master Thesis, Charles Rich and I [Rich & Shrobe, 1976] presented such a representation for abstract programs which we called *plans*. We reasoned that programs like other engineered artifacts should have a simple underlying conceptual structure consisting of a decomposition into parts and means for communication between these parts. When we specialized this observation to programs, we observed that the functions performed by programming language primitives fall into two categories which might be called "actions" and "connective tissue". Actions are modules which operate on a set of input data objects, yielding a set of new and modified output objects. Connective tissue arranges the flow of data and control between the actions.

For Complex Program Understanding

We then designed a formalism based on this observation. The formalism consists of segments, control-flow links, data-flow links, and abstract data objects. An abstract program is represented as a set of segments connected by data and control-flow links which specify how information propagates between the segments and which partially constrain the execution of the sub-segments. Segments are actions; they are used to represent the sub-steps of the program. Segments may be nested one within the other yielding a super-segment and sub-segment relationship.

Each segment has a set of local names for its input object and a second set of local names for its output objects; these names may be thought of as "ports". A Data-flow link is a directed connection between the ports of two segments. Typically the connection is made between the output port of one sub-segment and the input port of a second sub-segment, indicating the output object named by the first sub-segment's port will flow to the input port of the second sub-segment. A dataflow link may also connect the input port of a super-segment to the input port of one of its sub-segments; finally a dataflow link may connect the output port of a sub-segment to the output port of its super-segment. Data-flow links imply an ordering of execution; a segment which terminates a data flow link cannot begin execution until the datum is available at the initiating port of the segment. Control-flow links are directed connections between two segments, implying that the first segment must terminate before the second segment may begin. A plan consisting of segments and these two types of flow links may not completely constrain the ordering of sub-segment execution. Thus, as observed in [Sacerdoti, 1975] plans are non-linear. They are inherently a two dimensional structure the linearization of which accounts for most of the complication of language design.

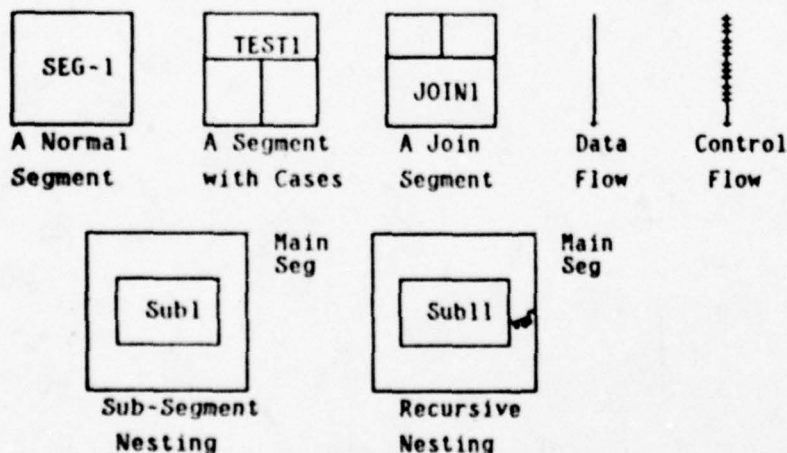
The plan formalism is intended to represent designs; however, these designs eventually turn into code in some particular language. A technique called surface flow analysis was developed to bridge the gap between the two forms of analysis. Primitives such as IF-THEN-ELSE, WHILE, variables assignment, argument passing, etc. which are concerned solely with ordering and communication are translated into data and control flow links. Other primitives such as arithmetic operations, CONS, CAR, CDR, etc. are translated into segments. Such surface flow analyzers have been developed for LISP [Rich & Shrobe, 1976] and FORTRAN [Waters, 1978].

During design a segment represents one step of a problem decomposition. Therefore, a means is required to specify abstractly what a segment does. This is done by stating the segment's *specs* which consists of: (i) a set of input names (ii) a set of preconditions which must hold immediately prior to program execution (iii) A set of output names (iv) A set of post-conditions which are guaranteed to hold immediately following the segment's execution. Alternatively, one can specify what a segment does by stating its plan, i.e. by presenting its decomposition into sub-segments.

Segments may have a conditional structure which is stated by breaking the segments up into cases. Each case is applicable under certain circumstances which are stated in the segment's specs. Control flow links can be attached to a particular case of a segment; the segment which terminates such a link is executed only if the particular case is applicable. This creates mutually exclusive control paths which can be united by a *join* segment.

The plan formalism can be interpreted by a symbolic evaluator which is in many regards quite similar to a LISP interpreter. However, the symbolic interpreter uses typical or symbolic data as input. Therefore, it must explore all control paths. In addition it must use a reasoning system to deduce whether the pre-conditions of each segment are satisfied. The symbolic evaluator is described in chapter 6.

I will often present plans using a graphical formalism. The symbols in this formalism are shown below.

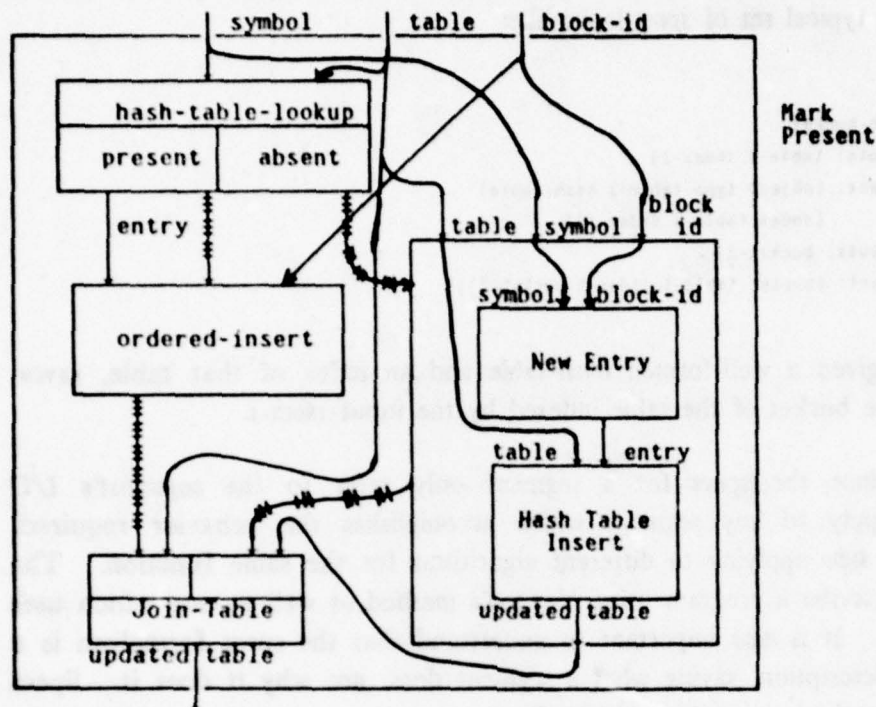


For Complex Program Understanding

As an example of how this formalism is used consider a programmer designing the symbol table for a block-structured language like ALGOL-60. A hash table might be used to store and retrieve the symbols efficiently. Each symbol is given a new entry in the table when it is first encountered; as the symbol is encountered in new blocks, the entry is marked with the BLOCK-ID of the new block.

To achieve a simple action such as marking a symbol with a block identifier (BLOCK-ID) several other operations such as HASH-TABLE-LOOKUP, SORTED-INSERT, RPLACA, etc. are called upon. These sub-actions interact to achieve the desired goal of having the symbol table indicate that the specified symbol is defined in the indicated block. This is done as follows: First, HASH-TABLE-LOOKUP is called to see if the symbol is defined in the table. If it is, the entry returned by HASH-TABLE-LOOKUP is passed to ORDERED-INSERT which inserts the BLOCK-ID of the specified block into the entry's list of BLOCK-ID's. If the symbol has no entry in the table, NEW-ENTRY is called to build a new hash-table entry; the new entry is created with a BLOCK-ID list including exactly the specified BLOCK-ID. This new entry is passed to HASH-TABLE-INSERT, which inserts it into the table.

This can be diagrammed as follows:



Plan Diagram For Mark Present Operation

HASH-TABLE-LOOKUP has a case structure; it performs a test and splits control into several paths, depending on the result of the test. The two control paths are rejoined by the join segment **JOIN-TABLE**. Notice that crossed lines show the flow of control between segments; normal lines show the flow of specific data objects.

Notice that many of the modules used to build **MARK-PRESENT** will eventually have internal structure of their own. **ORDERED-INSERT**, for example, will probably consist of a **SEARCH-LOOP**, a **CONS**, and a **RPLACD**. The hash-table routines will involve steps such as **HASH**, **BUCKET-FETCH**, etc. Thus, the structure given above is a layered one nesting boxes within boxes until one finally reaches programming language primitives.

Each segment in the plan above can be thought of as promising that certain conditions will hold after its execution as long as its preconditions are satisfied. Such sets of promises are stated using the specs formalism. As mentioned above, specs have four clauses. Two of these, *inputs* and *outputs* provide a list of input and output names which are bound to the actual inputs of the segment. The other two clauses

For Complex Program Understanding

36 An Engineering Theory of Evolutionary Design

are the *expect* and *assert* clauses which state the pre-conditions and the post-conditions of the segment. A typical set of *specs* looks like:

```
(defspecs fetch-bucket
  (Inputs: table-1 index-1)
  (Expect: (object-type table-1 hash-table)
            (index table-1 index-1))
  (Outputs: bucket-1)
  (Assert: (bucket table-1 index-1 bucket-1)))
```

which states that, given a well-formed hash-table and an index of that table, *FETCH-BUCKET* will return the bucket of the table indexed by the input *INDEX-1*.

Notice that since the specs for a segment only refer to the segment's I/O behavior, it can apply to any segment which accomplishes the behavior required. Thus, a spec is a *type* applying to different algorithms for the same function. The square root specs describe a program using Newton's method as well as one which uses the halving method. It is also important to understand that the specs formalism is a local and *intrinsic* description, saying what a segment does, not why it does it. Specs have no notion of method or purpose within them.

However, for an engineered device to function properly, it is necessary that the pattern of interactions between sub-modules guarantees that every module's expectations be satisfied at the time of its invocation. Further, the pattern of interactions must guarantee that the desired behavior of the whole device will result from the behaviors of the parts. It is only within this more global and *extrinsic* description that a notion of purpose is found. For example, in a hashing system we can talk about the purpose of the hashing step: it computes the index of the bucket in which the desired object should be found, eliminating the need to search through other buckets which cannot contain the object. Similarly, in a *HASH-TABLE-INSERT* routine the purpose of the *LIST-INSERT* routine is to splice the element into the appropriate bucket so that it will be a member of the table.

A plan consists of a pattern of sub-segments connected together by data and control flow links. Two kinds of requirements are found in a plan. First there are the requirements that each sub-segment's expect conditions must be satisfied; this is called a *pre-requisite* requirement. Second is the requirement that the overall goals of the main segment must be satisfied; this is called an *achieve* requirement. The first of

these requirements is indicated by the expect clauses of the sub-segment's specs; the second is indicated by the assert clauses of the main segment's specs. If the plan represents a reasonable design then it is possible to show how the behavior of the sub-segments interact to satisfy these requirements. It is possible to summarize such an argument so that it only refers to basic units of description, the spec clauses of those sub-segments involved in guaranteeing that the requirement is satisfied. These summarized arguments are called *purpose links*.

Consider the following diagram for a HASH-TABLE-INSERT routine. HASH is called with the table and the object to be inserted as arguments and calculates an index of the table. Fetch bucket is called with this index and the table, producing a bucket of the table; the bucket must be a linked list. Finally, the bucket and the object are passed to LIST-INSERT which side-effects the list, inserting the object into the list. This causes a derived side effect to the table; since one of its parts is side effected, the table is as well. The updated table is returned as an output of HASH-TABLE-INSERT. The pre-requisite and achieve conditions are indicated on the side of the diagram.

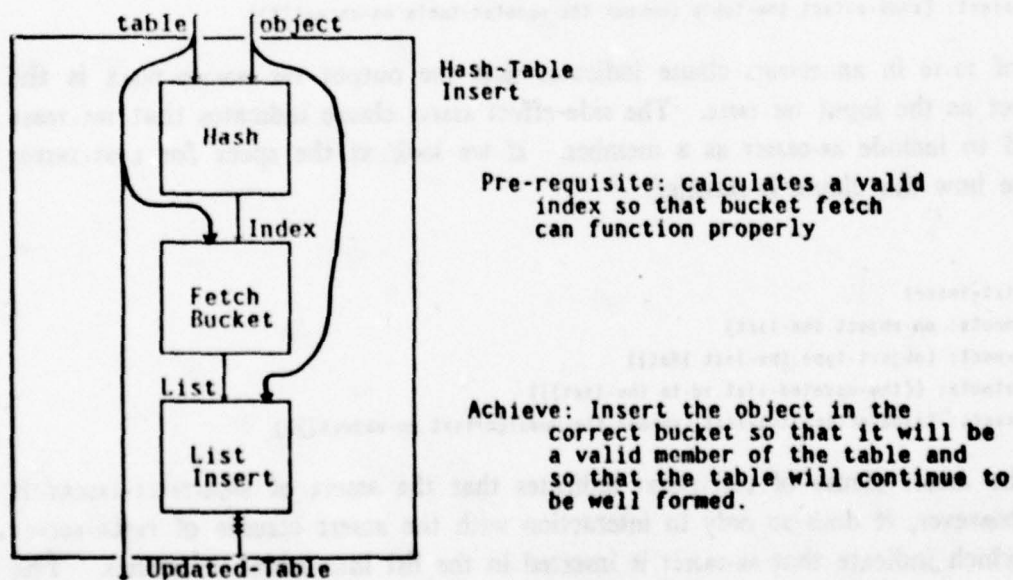


Diagram For Hash-Table Insert Routine

We can look at the specs of the sub-segments to see how the purpose links are developed. The specs for HASH are:

```
(defspecs hash
  (inputs: the-table the-object)
  (expect: (object-type the-table hash-table))
  (outputs: the-index)
  (assert: (object-type the-index number)
            (index the-table the-index)))
```

We have already seen the specs for *FETCH-BUCKET* above. Notice that the second *assert* clause of *HASH*, implies that the second *expect* clause of *FETCH-BUCKET* is satisfied. Now let us look at the specs for *HASH-TABLE-INSERT*

```
(defspecs hash-table-insert
  (inputs: an-object the-table)
  (expect: (object-type the-table hash-table))
  (outputs: ((the-updated-table id-to the-table)))
  (assert: (side-effect the-table (member the-updated-table an-object))))
```

The use of *ID-TO* in an *OUTPUTS* clause indicates that the output *THE-UPDATED-TABLE* is the same object as the input *THE-TABLE*. The side-effect *ASSERT* clause indicates that *THE-TABLE* is changed to include *AN-OBJECT* as a member. If we look at the specs for *LIST-INSERT* we will see how this clause is satisfied.

```
(defspecs list-insert
  (inputs: an-object the-list)
  (expect: (object-type the-list list))
  (outputs: ((the-updated-list id-to the-list)))
  (assert: (side-effect the-list (member the-updated-list an-object))))
```

Clearly, the *assert* clause of *LIST-INSERT* indicates that the *assert* of *HASH-TABLE-INSERT* is satisfied; however, it does so only in interaction with the *assert* clauses of *FETCH-BUCKET* and *HASH* which indicate that *AN-OBJECT* is inserted in the list into which it hashes. The satisfaction of the *assert* clause of *HASH-TABLE-INSERT* depends on *assert* clauses from each of these sub-segments.

We call these logical links between sub-segment behavior *purpose-links*, those links which explain how a sub-module's expectations are met are called *pre-requisite* links, those which explain how the overall intentions of the main segment are met are called *achieve* links. The pattern of purpose links, together with the data- and control-flow links, and the various sub-segment's specs is what we term a *plan*. Plans play a central role in the work of the programmer's apprentice because they explain the teleological structure of the program: the reason why each module is present and the logic of how the modules' configuration achieves the overall goal.

The addition of purpose links transforms the plan formalism from an abstract programming language to a design representation which includes not only a set of actions to be performed but also a statement of their teleological structure. Since the sub-segments of a plan may be specified at a high level of abstraction it turns out that the plan formalism can easily represent abstract teleological structures. As I've mentioned, there is a craft discipline among programmer's consisting of a repertoire of standard methods for achieving certain types of goals. There are standard ways to traverse a tree or a list structure, and standard methods for accumulating items into a set. These standard methods can be conveniently represented as *standard plans*, using the abstraction powers of the plan formalism to capture the significant generalities of a programming domain. The plan formalism has the added virtue of representing these techniques in a manner which is independent of the particular programming language being used.

Let me explain this a bit more before going on. Suppose I had a set of objects represented by some data structure and I wished to build a collection of all members of this data structure which satisfy some criterion. One standard technique for accomplishing this is what I term the *filtered-accumulation* plan. This plan consists of three sub-plans. The first is an *enumeration* plan which generates the elements of the original data-structure; if this data structure is a list then this plan would have a familiar pattern of "cdring down" the list; if the data structure is a binary tree, the enumerator would have the structure of "car cdr" recursion. The second sub-plan is a *filter* plan which tests the elements produced by the first sub-plan selecting those which satisfy the criterion. The final sub-plan is an *accumulation* plan which builds a new data structure containing those elements which passed through the filter. If the final data structure is to be a list, this sub-plan would have the familiar pattern of "consing up" a list.

40 An Engineering Theory of Evolutionary Design

Now consider the code for two versions of this idea. In the first version (written in ALGOL) the original and final data structure are arrays; the second version (in LISP) uses a binary tree and a list.

```

integer array A[0:100], B[0:100];
integer i,j;
j := 0
for i := 0 step 1 until 100 do
  if Criterion (A[i])
    then begin
      j := j + 1; B[j] := A[i]
    end

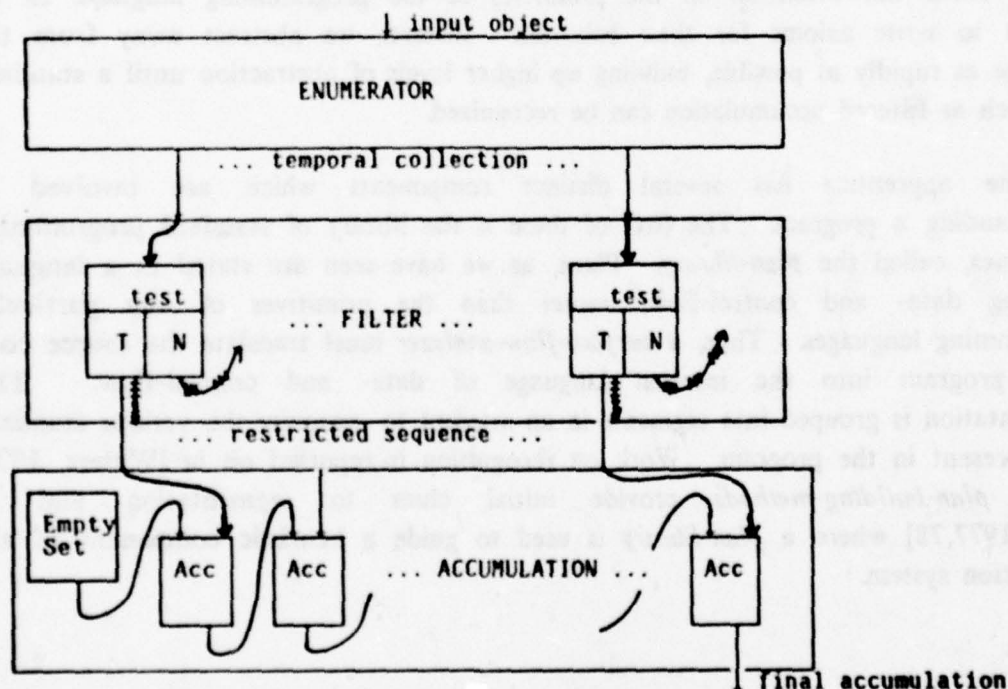
(defun fil-acc (tree) (fil-acc-1 tree nil))

(defun fil-acc-1 (tree acc)
  (cond ((Criterion (value tree)) (setq acc (cons (value tree) acc))))
  (cond ((terminal tree) acc)
        (t (fil-acc-1 (left tree)
                        (fil-acc-1 (right tree) acc))))))

```

Even ignoring the language differences, there is clearly quite a bit of difference between the two programs, yet I have already claimed that they are actually instances of the same general technique, *filtered-accumulation*. The plan formalism captures this generality using *temporal abstraction*. Temporal abstraction looks at the history of the computation, grouping together occurrences of segments of like type. For example, in the LISP program there are recursive invocations of `FIL-ACC-1` producing several occurrences of segments of this type. Similarly in the ALGOL program the loop executes repeatedly producing several occurrences of the loop. Temporal Abstraction aggregates all these occurrences into a single new, abstract segment which is called the *enumerator*. Each of the occurrence within the enumerator produces an output object. In the ALGOL program the output is the contents of the *i*th array slot; in the LISP program, the output is the value part of the current tree node. These outputs are aggregated into a new, abstract data structure called a *temporal collection* (since the objects are produced one by one, the collection exists across time, rather than as a single unified data-structure).

We may similarly observe that in each program there are repeated occurrences of the *CRITERION* test. These may be aggregated into the *filter* sub-segment. We may note that each program has a repeated accumulation step. In the ALGOL program this consists of the two steps of adding one to *j* and then storing a quantity into the *j*th slot of the array *a*; in the LISP program the accumulation is performed by the *cons*. Again, the repeated occurrences of these steps can be aggregated into a segment. Once this is done, we can notice that the filter segment will contain a number of identical test segments which have no data flow between them. However, from the successful case of each test segment there is a data flow to a sub-segment of the accumulation plan. Internally, the accumulation plan is a *cascade* of identical *set-accumulators*, each of which takes two inputs: (1) a set which is input from the previous *accumulator* and (2) a new element; the accumulator produces a new set which includes all the previous elements plus the new one. The set output by the final accumulator is the output of the whole filtered-accumulation plan. From this viewpoint both programs have the following common structure.



Temporal View of Filtered Accumulation

Notice that at this level of description we have left many features unspecified. For example, we have not said what type of object is input to the enumerator nor how it works internally. In spite of this, we do know that this pattern of interactions (i.e. this *plan*) produces a set whose members are a subset of the elements contained in the enumerated object. Furthermore, we know that this subset consists of exactly those members which satisfy the criterion of the filter. Indeed, this general pattern of segments is so common that one ought to *recognize* it where it occurs and immediately infer that the output is exactly this subset. Languages such as CLU [Liskov, 1974,77] and ALPHARD [Wulf, 1974,76] have introduced *iterators* and *generators* to make it easier to capture these and similar notions. Temporal abstraction will be discussed in detail in chapters 8 and 9.

The Apprentice approach to program understanding is distinct from the approach of program verification systems like [King, 1969], [Deutsch, 1973], [Igarashi et al., 1975]. In the Apprentice, although we require the usual logical techniques we do not focus our attention on the primitives of the programming language in an attempt to write axioms for their behavior. Instead, we abstract away from the language as rapidly as possible, building up higher levels of abstraction until a standard plan such as filtered accumulation can be recognized.

The apprentice has several distinct components which are involved in understanding a program. The first of these is the library of standard programming techniques, called the *plan-library*. Plans, as we have seen are stated in a language involving data- and control-flow, rather than the primitives of any particular programming languages. Thus, a *surface-flow-analyzer* must translate the source code of a program into the internal language of data- and control-flow. This representation is grouped into segments in an attempt to *recognize* the various standard plans present in the program. Work on recognition is reported on in [Waters, 1978] where *plan-building-methods* provide initial clues to segmentation and in [Rich, 1977,78] where a *plan-library* is used to guide a heuristic component of the recognition system.

The recognition systems, however, must call on a reasoning system from time to time to see whether their proposed recognition of the code is feasible. In later chapters as I present the deductive component more carefully we will see the use of a *task-agenda* and an explicit recording of *dependencies*. These are the methods by which the deductive component communicates with other parts of the system.

The deductive component of the system plays a second role in the apprentice which I refer to as *plan-verification*. The apprentice requires a large library of standard plans whose properties have already been analyzed and recorded. While it is a theoretical possibility that such complete analyses could be produced by hand, in practical terms this is prohibitive. Instead, the deductive component of the system is used to show that a plan (stated at any level of abstraction) satisfies certain properties. In this use, REASON is presented with a *plan-diagram* consisting of data- and control-flow links and specifications for the sub-segments used in the diagram. It is then asked to show that certain properties hold; often it is useful to give REASON a set of lemmas to be proven first which will structure the proof and make it more comprehensible. REASON is also allowed to ask for help if it feels that it is getting lost. As the system develops the proof, it records all its deductions. These are then summarized into the pre-requisite and achieve links of the plan which is filed in the plan library as a new standard plan.

In the typical interaction with the apprentice, as seen in the scenario, the programmer first develops a design for a segment of code, using the plan library as a shared vocabulary of high-level building blocks. As these pieces are woven together, the apprentice checks that pre-conditions of each segment are satisfied and warns the programmer of design bugs if any precondition is violated. When the programmer believes that a total plan has been formulated he asks the apprentice to check whether this plan does achieve the intended goals.

In general the programmer will not begin to code a segment until he has gone through this design-checking protocol with the apprentice. Having completed the design at the level of abstract plans, however, he goes on to write the code. It is at this point that the surface analyzer and the recognition components are called on to match the code and the already verified plan. If the alignment is made, then the programmer proceeds knowing that his program accomplishes the things which he had asked the apprentice to check.

Frequently, however, the programmer may find it more convenient to write the code without going through the design protocol. Under these conditions the apprentice will have weaker clues and will have to interact with the programmer more often, asking for specifications and other hints to guide its analysis. In any event, once the analysis is complete the apprentice will have constructed a recognition mapping between the code and the plan for the segment; in addition most of this plan will have pointers back to plan fragments from the library. Thus, the apprentice can explain the code using the high-level vocabulary of the library. Furthermore, the apprentice will have developed and written in the notebook a complete explanation of the intermodule dependencies, giving it the ability to examine how changes to one of the sub-segments will affect the behavior of the whole program.

In summary, the plan of an engineered device is a set of logical connections between the conceptual descriptions of sub-modules, the descriptions of implementation strategy, and the overall intentions for the device being engineered. These logical steps explain how each module of the overall device contributes to the higher level conceptualization as well as why each sub-module is capable of functioning. The lack of such a logical connection in a proposed device would indicate a conceptual failure or design bug. Since any engineering discipline builds up a repertoire of standard plans, understanding an engineered device is largely a matter of recognizing which standard plans are used and how they are interfaced to achieve their intended goals.

Given that modules of a device may themselves be conceptual constructs with internal structure, plans provide an abstracting mechanism describing the structure of the device at a level appropriate to the task at hand. Plans also allow one to describe and reason about the behavior of incompletely designed devices, since a module's net behavioral specifications may be used within a larger plan even if there is as yet no internal plan to accomplish the behavior of the sub-module.

Section 2.7: Plans in Maintenance and Explanation

Plans, as outlined above, give a teleological description of program behavior, abstracted to a level of description which is convenient to the programmer. It is a rather trivial matter to generate explanations of a program from a plan. Since plans contain more information than does the program itself, such explanations will be richer than a mere recitation of the code.

My goal, however, is to understand and support the process of program evolution. As I have noted, plans capture the relationship between program design choices, abstract modularization, and overall intentions. In doing this, they localize the effects of a change in design strategy, and specify the teleological requirements which must be satisfied in any modification of the design.

As a simple case, consider a hash-table insert routine which has been implemented using ordered linked-list buckets with a count field. The code for such a program might be:

```
(defun insert (item)
  (insert-in-bucket (table (hash (key-part item))) item))

(defun insert-in-bucket (bucket item)
  (do ((previous-list bucket (cdr previous-list))
      (current-list (cdr bucket) (cdr current-list)))
      ((null current-list)(rplacd previous-list (list item)))
      (and (greater-than (car current-list) item)
           (rplacd previous-list (cons item current-list))))
  (rplaca bucket (1+ (car bucket)))))
```

Suppose that for space efficiency we decide to change to a rehashing scheme. Since this change is strictly a design issue dealing with buckets, the plan tells us that the overall structure of the insert module itself will not have to change, but that the Insert-In-Bucket module as well as the communication between the two modules might require change. It further tells us that the last line (i.e. the `rplaca` which bumps the count) is no longer relevant. At first glance one might guess that this is all the help one could get.

However, the plan library reveals that there is more structure in common between the old and new designs. In the library, plans and data-structures are organized into (tangled) hierarchies where objects lower in the hierarchy inherit properties from those above them. In both implementations of a hash-table we have that the buckets are *linear-objects*; furthermore, we have a generalized version of the *search-loop*, called *linear-search-loop* which can search any *linear-object* such as lists or arrays. The more specific versions of linear-search-loop differ only where the choice of representation for the particular linear-object is relevant.

This difference appears in the BUMP, EXHAUSTION, and TERMINATION steps. In the *re-hash* scheme, BUMP is the RE-HASH operator and successful termination of the search is indicated by a special marker (such as nil) indicating that a slot is free. Exhaustion of the search might be indicated by the RE-HASH routine returning a negative number. An item is made a member of a bucket in the RE-HASH scheme by inserting it in the array.

In the *linked-list* version, BUMP is the CDR operation; objects are selected by CAR, exhaustion of the list is indicated by the presence of NIL. Successful location of a place to insert the object is indicated by the presence of a larger element in the next position. Using this information, the system guides the programmer to the following new program (I will discuss this idea further in chapter 13):

```
(defun insert (item)
  (insert-in-bucket (hash (key-part item)) item))

(defun insert-in-bucket (initial-slot item)
  (do ((slot initial-slot (rehash slot)))
      ((minusp slot)(error 'no-slots-left))
      (and
       (null (table slot))(store (table slot) item))))
```

Typically, the apprentice builds more than one viewpoint of the program during the recognition process. During program modification, one or another of these viewpoints might provide a perspective from which the effect of the modification appears quite localized. In any event, since the apprentice has a complete record of all the logical dependencies, it can easily evaluate whether any proposed modification can damage a desired property.

Section 2.8: Dependency Directed Reasoning

A plan may be thought of as an abstract program coupled with a logical analysis. However, it is important to note that this logical analysis need not necessarily be a "proof" in the sense of a guarantee of correctness. REASON is capable of conducting logical arguments which range from the informal or "common sense" to the rigorous. In many cases the plan for a program will only contain a "common sense" or engineering analysis which is inadequate to guarantee correctness under all conditions, but which is good enough for purposes of explaining its teleological structure. When necessary, REASON can be asked to verify certain modules and can carry this out with full rigor. We often observe experts making an analysis in exactly this way: First they conduct a common sense analysis to explicate certain facts and to establish a framework of understanding; once this is accomplished the framework guides a more formal analysis, keeping it from getting lost in a sea of combinatorics. It would be desirable for REASON to be able to do something like this.

Another desideratum is that an incremental change in the program should necessitate only incremental changes in the analysis of the program. To partially meet these desiderata REASON was designed as a dependency based system. In a dependency based system every new assertion entered into the data base is accompanied by a *justification* stating which other assertions form the logical support for the new one. The justification itself is an object which the system can inspect and manipulate.

Assertions in the reasoning system have one of two statuses: *in* or *out*. An *in* assertion is one which is believed. An *out* assertion is one not currently believed. A special module called the Truth Maintenance System (TMS) [Doyle 1978] is responsible for guaranteeing that all assertions with valid reasons to be believed are *in* and all assertions lack valid justifications are *out*.

REASON has several uses for dependency based reasoning: management of abstract models for programs, analysis of program modifications and hypothetical reasoning during theorem proving. In chapter 12 I will discuss the use of dependency based reasoning in the the analysis of side-effects. In this situation, REASON first conducts a simple analysis assuming that the degree of sharing between complex data structures is limited. Various desired properties of the program are then proven, under this assumption. Sometimes such a cursory analysis is sufficient. However, when a more careful exploration is desired, the assumption can be removed and

For Complex Program Understanding

replaced by a more cautious assumption or by no assumption at all. The TMS uses the dependencies to determine what conclusions remain valid under the new assumptions. In many cases, some of the important properties of the program do not depend on the assumption and remain *in*. However, if some property does in fact depend on the assumption it will go *out* indicating that the original proof is not still valid under the conditions of sharing. A more detail proof can then be attempted.

I will now turn to the detailed presentation of the techniques used in REASON. In chapter 3 I will first discuss the reasoning system per se; chapter 4 will introduce the task agenda and the system's method of explicit control. Chapter 5 will present the program description techniques in detail. Chapter 6 presents the symbolic interpreter for the plan formalism and chapter 7 gives an example of how this can be used in program verification. This chapter is quite tedious and can be skipped without loss of continuity. Chapters 8 and 9 detail the techniques of temporal abstraction and its use in analysis by inspection. In chapter 10 a language for describing data structures is introduced along with a catalogue of data descriptions which REASON uses. These descriptions are used during program analysis and recognition and are important to the material which follows on the analysis of side effects. However Chapter 10 need not be read very carefully to understand the material which follows. Chapter 11 presents my techniques for reasoning about side effects by making simplifying assumptions. This material is extremely novel and quite distinct from verification literature on the same subject. Chapter 12 is a brief discussion of some concepts which can be used to make the ideas of chapter 11 more powerful. Finally, in chapter 13 many of the previous ideas are combined in a sketch of how REASON will eventually be able to support program evolution.

Chapter 3: The Reasoning System

Understanding programs requires a sophisticated reasoning capability. This chapter describes REASON's basic deductive system. REASON has its antecedents in two separate works. The first of these is an earlier program implemented in LISP which was reported on in [Rich & Shrobe, 1976]. The current version of REASON is written in a variant of AMORD [DeKleer, et. al. 1977] a language for constructing problem solvers. Both systems maintain a dependency network, but the AMORD system does so in a cleaner manner, utilizing the Truth Maintenance System [Doyle, 1978]. I will begin by reviewing the basic concepts and constructs of the system.

Section 3.1: Dependencies and Justifications

REASON is implemented in a variant of the language AMORD. I will begin by reviewing the syntax and basic concepts of this language.

Imagine a reasoning system which knew that numerical ordering is transitive. Suppose also it knew that X was less than Y and that Y was less than Z. Presumably, an "ordinary theorem prover" would then conclude that X was less than Z. However, the system could in principle deduce more than just this. It knows that X is less than Z *because* it is less than Y which is in turn less than Z and *because* the ordering is transitive.

REASON like some other newer systems [Doyle, 1978], [London, 1977], [Stallman and Sussman, 1977] regards the justification for the new fact as an object of great importance to the theorem prover itself. The justification for the new fact tells us what other facts the new fact depends on. If we did not believe that X was less than Y or that Y was less than Z or that numerical ordering is transitive then we ought not to believe that X is less than Z. A justification states such a dependency between facts.

REASON's goal is not only to prove properties of a program but to understand how these properties follow from known or assumed properties of sub-modules. Hence such dependencies are a crucial form of information in REASON. When an assertion is entered into REASON's data base, it is always accompanied by a justification explaining why the new assertion is believed. To make this convenient, as each

For Complex Program Understanding

50 The Reasoning System

assertion is entered into the system's database it is assigned a unique "fact-name" by which it may be referenced. For example:

Assertion	System-Supplied Fact-Name
(< X Y)	F-1
(< Y Z)	F-2

The user may add the the fact deduced from the above by calling the **ASSERT** function:

```
(Assert '(< X Z) '(Transitivity F-1 F-2))
```

ASSERT takes two arguments: the new assertion to be added to the data base and the justification for the assertion which is a list whose first element is a justification type and whose remaining elements are the fact-names upon which the new assertion depends.

```
(Assert <fact> (<justification-type-name> ... <fact-name> ...))
```

One important justification type is **PREMISE** which, as the name suggests, indicates that the fact is believed without further justification. A **PREMISE** justification has no supporting facts. The three facts about ordering shown above could well have been entered into the system as follows:

User Types	System-Supplied Fact-Name
(Assert '(< X Y) '(Premise))	F-1
(Assert '(< Y Z) '(Premise))	F-2
(Assert '(< X Z) '(Transitivity F-1 F-2))	F-3

The rudimentary facility of any logic system is a mechanism for making deductions. **REASON** accomplishes this using *rules* which consist of two elements: a *trigger-set* and a *body*. The trigger-set is a list of *patterns* each of which has two parts: a *fact-name-variable* and an *assertion-pattern*. The body is a LISP expression which is evaluated in an environment in which the variables of the patterns are bound to the objects which they match. The following is a fairly typical **REASON** rule:


```

(rule ((:f (Rest :list-1 :list-2))      trigger
      (:g (Member :list-2 :obj-1)))    set
      (assert '(member :list-1 :obj-1) '(List-Membership :f :g)))
      fact                             justification

```

Variables are indicated by a leading colon (:).

Here the body is the assert statement. The trigger set is the list:

```

((:f (Rest :list-1 :list-2))
 (:g (Member :list-2 :obj-1)))

```

In these triggers, the leading single variable (:f or :g) is the fact-name variable, the remaining part of each trigger ((REST :LIST-1 :LIST-2) OR (MEMBER :LIST-2 :OBJ-1)) is the assertion pattern. Rules are dealt with in 3 stages.

- (i) When an assertion is added to the data base, all rules with a trigger whose assertion pattern matches the new assertion are *triggered*.
- (ii) Each of the remaining triggers are examined to see if their assertion pattern also matches an assertion in the data base. However, these matches must be consistent with the variable bindings created by the earlier matches.
- (iii) If all of the triggers have a matching assertion, then the rule is *applicable* and its body is executed in the binding environment created by the match.

As each trigger is matched to an assertion, the fact-name variable of that trigger is bound to the fact-name of the matched assertion. This allows the body of the rule to refer to its triggering facts. In particular, assert statements in the body of the rule may include a justification mentioning these facts.

At each moment any assertion has one of two statuses in REASON, it is either *in* or *out*. A fact which is *in* is believed to be true. An assertion whose negation is *in* is believed to be false. If both an assertion and its negation are *in* then the data base is contradictory and corrective action is required. If neither the assertion nor its negation is *in*, then the truth-value of the fact is simply unknown.

52 The Reasoning System

assertion	negated assertion	meaning
in	out	assertion true
out	in	assertion false
in	in	contradiction
out	out	truth value unknown

Facts which are *out* may be brought *in* by asserting the fact with a valid justification. A fact which is *in* may be made *out* using the function **RETRACT** to remove a valid justification. If no valid justification is left, the fact goes *out*. When a fact goes *out* a check is made to see which other facts are affected by the change of status of the first fact. If the first facts going *out* invalidates the support of some other fact, then the same checks are made recursively, outing all facts which now lack well founded support. Similarly, if a fact comes *in* because new support for it is discovered, then a check is made to see which other facts are affected. Any facts whose support is made valid by a change in status of the first fact are brought *in*. Such *ining* and *outing* of facts is managed by the Truth Maintenance System, which insures that only facts with well founded support are *in*. Thus, a fact which has never been asserted, is by definition *out*.

The meaning of justifications such as the transitivity justification shown above is that whenever *F-1* and *F-2* are *in* then *F-3* should also be *in*. If for some reason either *F-1* or *F-2* became *out*, then *F-3* would lack support and would also become *out* (unless it has other support).

It is frequently necessary to *assume* that some fact holds even though no reason exists for believing the fact. This is often done in hypothetical reasoning as when one proves that *A* implies *B* by assuming *A* and deriving *B*. Since the assumed fact *A* has no simple support (as for example ((*x* *z*)) above) a different type of justification is required. One assumes a fact by making it depend on the *outness* of its negation; thus, if the negation of the assumed fact should ever be proved, the assumption will go *out*. This is done by the function **ASSUME**:

```
(Assume '(Made-of The-Moon Green-Cheese) '(Bill-said-so F-23))
```

which states that the system will believe that the moon is made of green cheese as long as it has no reason to believe that the moon is not made of green cheese and as long as it believes fact F-23 (which presumably is a statement of what Bill told the system). ASSUME takes two arguments: a fact to assume, and a list whose first element is an assumption-type-name (used only for mnemonic value) and whose *cor* is a list of fact-names which indicates the reasons for making the assumption. Whenever all the fact-names in this list are *in* and the negation of the assumption is *out*, the assumed fact is brought *in*.

This requires the justification built by ASSUME to have two parts: A list of assertions upon whose *inness* the fact depends and a list of assertions upon whose *outness* the fact depends. When the above ASSUME form is invoked it creates the assertion:

(Not (Made-of The-Moon Green-Cheese)) F-1001

and then justifies the assertion

F-1002 (Made-of the-moon green-cheese)

by stating that F-1002 depends on F-1001's *outness* and on F-23's *inness*.

F-1002 (Made-of The-Moon Green-Cheese) (Bill-said-so (F-23)(F-1001))

The justification of F-1002 is not the same list as the second argument of the ASSUME which created F-1002. The justification is built by ASSUME using the information provided in its calling arguments. The first list in the justification is the list of facts whose *inness* supports F-1002 and the second list is the *out* list. This support structure, which was originated in Doyle's TMS [Doyle, 1978], allows the system to believe that the moon is made of green cheese until some deduction provides valid support for F-1001. At that point, F-1001, the negation of F-1002, will become *in* (and thus not *out*). Since F-1002 depends on the *outness* of F-1001 it will lack support and thus become *out* itself. Similarly, if other facts had been deduced from F-1002 they would now lack support, since F-1002 would no longer be *in*. The Truth Maintenance system propagates these changes of status until only assertions with well-founded support remained *in*.

Notice that in the support structure for assumptions, adding a new assertion to the data base (e.g. the negation of an assumed assertion) can cause an assertion which is *in* (the assumed one) to become *out*. Thus, the number of things believed to be true can decrease as assertions are added. For this reason, such a support structure is referred to as *non-monotonic*. When a contradiction is detected, the system finds those assertions which are supported by non-monotonic dependencies (i.e. on the *outness* of others) and brings *in* one of the assertions upon whose *outness* they depend. Since the system maintains the dependencies between facts, it is easy for it to find only those assumptions which logically are related to the contradiction and to use these as the candidates for rejection. This avoids the thrashing which was found to occur in chronological backtracking systems such as Micro-Planner [Sussman, et. al. 1971] and even in more flexible systems where dependencies were not explicitly maintained. This process is called *dependency-directed backtracking* [Stallman & Sussman, 1977].

Sometimes it is necessary to make an assertion depend on the *inness* of some facts and the *outness* of a second set of facts. This can be done by calling ASSERT with a justification argument whose first element is SL. Such a justification should have three other elements: A mnemonic name, a list of facts upon whose *inness* the new fact depends, and a list of facts upon whose *outness* the new fact depends. For example:

```
(Assert '(Hacker Howie) '(SL MIT-people-hack (F-1) (F-2)))
```

will create the new assertion *F-3* justifying it so that it will be *in* whenever *F-1* is *in* and *F-2* is *out*. Thus, we would have:

Fact-Name	Assertion	Justification
F-1	(Hacker Howie)	(MIT-people-hack (F-1) (F-2))

A final type of justification arises in the proofs of implications. As mentioned above, typically one proves (IMPLIES A B) by assuming A and deriving B. The justification of (IMPLIES A B), however, is not logically the justification of B. It is, instead, exactly those facts which were involved in deriving B from A which were do not themselves depend on A. For example, consider the following trivial proof:

Fact-Name	Assertion	Justification
F-1	(Implies A C)	(Premise)
F-2	(Implies C D)	(Premise)
F-3	A	(Assumption)
F-4	C	(Modus-Ponens F-3 F-1)
F-5	D	(Modus-Ponens F-4 F-2)
F-6	(Implies A D)	(Conditional-Proof F-1 F-2)

As I mentioned above, the logical support of F-6 is precisely F-1 and F-2. To calculate this the system can trace back through the dependencies to find those facts which support F-5, the consequent of the implication. These are F-4, and F-2. F-4 is, in turn, supported by F-3 and F-1. Of these assertions, we eliminate the hypothesis F-3 plus those assertions which depend on it. These are F-3 and F-4, leaving only F-1 and F-2 which are then the support of F-6. The system is instructed to perform such an analysis by asserting a fact with a justification whose justification-type is *Conditional-Proof*:

```
(Assert '(Implies A D) '(Conditional-Proof F-5 F-3))
```

ASSERT performs a special analysis when given a second argument whose first element is the justification-type conditional-proof. The first fact-name in a Conditional-Proof justification is the consequent of the implication, the second fact-name is the hypothesis of the conditional proof argument. ASSERT creates a justification for the above assertion in which the support for the assertion is the set of facts (such as F-1 and F-2) upon which the implication relies. Thus, if F-1 were to go *out*, F-6 would lack support and go *out* itself.

To facilitate the construction of complex dependency structures the system includes a primitive for creating an assertion which has no justification. This primitive is called ASSERTION and takes a single argument, the assertion to be created. It returns the fact-name of the new assertion. For example, evaluating

```
(Assertion '(Hacker Howie))
```

56 The Reasoning System

will cause a new assertion to be built and assigned a fact-name, say *F-2001*. Since this new assertion has no valid justification (it has no justification) it is *out*. At some later time, some rule might decide to give this fact a justification; if this justification is valid, *F-2001* will come *in*. For example, if we had the following fact and rule:

```
F-2002    (At-Mit Howie)

(rule ((:f (At-Mit :person)))
      (Assert '(Hacker :person) '(MIT-is-full-of-hackers :f)))
```

Then the system would assert

```
F-2001    (Hacker Howie) (MIT-is-full-of-hackers F-2002)
```

Remember that *F-2001* was originally created with no justification. Suppose at that point another assertion *F-2004* was created and made to depend on the *outness* of *F-2001*, as follows:

```
(let ((a (assertion '(Hacker Howie))))
      (Assert '(Is-Careful howie) '(SL Hackers-are-loose () (,a))))
```

[Note: the comma used above is an "unquote" which causes the variable *a* to be evaluated even though its inside a quoted form. Also *LET* is a macro defined in the standard MacLisp. The first argument to *LET* is a list of pairs of variables and forms; each form is evaluated in the enclosing environment and then each variable is bound to value of its corresponding form. The remaining arguments to *LET* are forms to be evaluated in the new environment created by the bindings.]

When *F-2001* was first created, it was *out*; therefore *F-2004* was *in*. However, when the rule above is triggered by *F-2002* it executes and brings *F-2001* *in*. Since *F-2004* depends on the *outness* of *F-2001* it then goes *out*. Conversely, if the support for *F-2002* is ever remove, using *RETRACT* for example, the *F-2001* will go *out* and *F-2004* will come *in*. This allows *REASON* to use justifications in building its control structures.

I will now turn to the issue of control within the reasoning system.

Chapter 4: Explicit Control and The Task Network

The traditional weakness of automatic deduction systems is that they are prone to blind searches. The room for exponential explosion is so large that even large amounts of a fixed factor overhead are justified if they can cut down the size of the search space.

The approach I have followed here is to represent all control of the deductive process explicitly in a form which can be manipulated by the same mechanisms as those which conduct the logical deductive process itself. Such an approach has been followed in [DeKleer, et. al. 1977], [McDermott, 1977]. This allows the deductive to be self-conscious, able to explain what it is doing and why it is doing it. Such a system can reason about whether it ought to continue to pursue a particular task, or rather abandon it as hopeless or of too little importance to command further resources and attention. A system which is explicit in its control discipline can exhibit flexibility which is precluded in more traditional systems which encode their control in the state of procedures which can not be examined.

This suggests a system which at the very least knows what task it is attending to and where that task fits into its larger goals. REASON organizes its operation around a data-structure called the *task-network* [McDermott, 1977] which makes this information explicit. The task network is represented by assertions in the data-base used to record facts about the program being analyzed. However, the control assertions in the data-base have a justification structure which *outs* them once their usefulness has passed.

A simple example of the use of control assertions in consequent reasoning will perhaps clarify the discipline used in REASON. In consequent reasoning the system attempts to chain backward from its current goal to sub-goals which interact to imply the main goal and which are (hopefully) closer to facts which are already known explicitly.

For any particular goal there might be several different methods for deriving the desired fact. A particular fact about a list might be derived by backward chaining though some implication, or it might be deduced by structural induction. In general one would tend to prefer the simpler method, however, there are cases in which the opposite would be the better strategy.

This has led to the following protocol. A goal is entered into the system by calling the primitive `GOAL-ASSERT`. This takes three arguments: The first of these is the assertion to be proven. The second argument is used to indicate what higher level task gave rise to this goal. (this is usually some sub-task of the symbolic evaluator, but for simplicity of presentation I will call this task *top-level*). The third argument is a justification (just as would be given to `ASSERT`). A goal created by `GOAL-ASSERT` should remain *in* as long as it is neither refuted nor satisfied and as long as its justification is valid. To achieve this, `GOAL-ASSERT` creates two assertions one stating that the goal is satisfied, the other stating that it is refuted. `REASON` builds an assertion stating the existence of the new goal. This assertion is given a justification which is identical to the justification passed in as an argument except that the new justification includes a dependency on the *outness* of the assertions which state that the goal is satisfied or refuted. The goal assertion will remain *in* until the goal is either satisfied or refuted at which time it will go *out*.

The assertion of a goal is an implicit request for the proposal of methods which might be capable of proving the assertion. If the particular goal is of a type for which a method is known, then the method is proposed. This proposal is given a justification which points to the goal assertion. A special procedure called the `ACCEPTOR` is responsible for choosing the order in which the various methods for a goal should be tried. The primitive `PROPOSE-METHOD` is used to propose a method; it takes three arguments: an assertion stating the method to propose, a justification for this assertion, and a body to execute if the method is ever accepted.

If the desired goal is ever proven, then an assertion is made saying that the goal is satisfied. If the negation of the goal is ever discovered, an assertion is added stating that the goal has been refuted. Either of these events causes the original goal assertion refuted to go *out*, taking with it all of the dependent control assertions. However, normal fact assertions will never depend on these control assertions; even when the control assertion are made to go *out*, the facts deduced stay *in* if they are logically valid.

What makes this protocol possible is a mechanism (developed in [Doyle, 1977]) which establishes well-defined points at which the system may chose which method to pursue. `REASON` is a queue based system whose main loop consists of finding pairs of rules and matching facts. At each iteration one such pair is removed from the queue and processed, potentially creating new facts and rules and thus new pairs of matching facts and rules. However, at certain times there will be no such pairs of

rules and facts to process. The method proposing and accepting protocol guarantees that new triggering pairs will not be created in an explosive manner, but will rather produce proposals for actions which must be accepted before new actions can occur. Even though the tree of possible methods and sub-goals might be exponentially explosive, the system has the option of choosing which branches to leave unexplored. As long as the system chooses to pursue only a few branches at a time, the queue will run out of pairs frequently.

This is the occasion for the special ACCEPTOR procedure to be invoked. The acceptor is a procedure run each time the queue runs out. Its purpose is to examine the network of goals and methods, hopefully finding at least one which it deems worth pursuing. The acceptor is allowed to add control assertions to the data base and these are allowed to trigger rules which will, in turn, add assertions to the data base. However until it accepts a method, no further work on the goals at hand will be performed. This organization, which is still being developed, allows the machinery of the reasoning system to be used in deciding which goals should be pursued. Once such a decision is made, a method is accepted triggering the rules which actually do the work.

A simple example will illustrate the technique. Suppose we want to prove *P* and we have (*IMPLIES Q P*), (*IMPLIES R Q*) and *R*. We would start off by stating that we have the goal *P*:

```
F-1      (Implies Q P)          (Premise)
F-2      (Implies R Q)          (Premise)
F-3      R                      (Premise)
(GOAL-ASSERT 'P '(top-level) '(Premise))
```

Since a goal statement has been entered, the system makes the assumption that the goal has as yet been neither satisfied nor refuted. Also it creates a goal statement for the newly created goal and justifies this statement as explained above.

60 Explicit Control and The Task Network

F-6	(Satisfied F-9)	(); no justification, therefore <i>OUT</i>
F-8	(Refuted F-9)	(); no justification, therefore <i>OUT</i>
F-9	(Goal P (top-level))	(Sub-goal() (F-6 F-8)) F-6 & F-8 <i>OUT</i> implies F-9 is <i>in</i> .

Notice that the goal assertion *r-9* includes both the fact to be proved and a list of the super-tasks which have led to the existence of this goal. Two rules are also created by GOAL-ASSERT; one watches for the goal becoming true, the other watches for refutations. Both these rules depend on the fact *r-a*.

R-1	(Rule ((:f P))	(Sat-Rule F-9)
	(Assert '(Satisfied F-9)	
	'(Satisfaction :f)))	
R-2	(Rule ((:f (Not P)))	(Ref-Rule F-9)
	(Assert '(Refuted F-9)	
	'(Refutation :f)))	

The assertion of the goal statement is an implicit request for the proposal of methods which might achieve the goal. The various method proposers now come into play. One obvious method is backward chaining, finding an implication whose consequence is the desired goal, and then posing the antecedent of the implication as a sub-goal. The following rule proposes the backward chaining method and then conducts the proof if the method is accepted.

```
R-10 (Rule ((:f1 (Goal :consequent :stack))
  (:f2 (Implies :antecedent :consequent)))
  (Propose-Method
    '(Method :f1 (Backward-Chain :f2)) '(B-C :f1 :f2)
    (goal-assert :antecedent '(:consequent . :stack) '(bc-sub-goal :f))
    (Rule ((:g (Implies :antecedent :consequent))
      (:h :antecedent))
      (Assert :consequent (Modus-ponens :g :h))))))
```

Notice the use of the primitive Propose-Method. This is actually a macro which expands as follows:

```
(Propose-Method Meth Just body) => (Let ((:a (Assert meth Just)))
                                     (Rule ((:b (Accepted :a)))
                                             body))
```

The method proposer above leads to the following results:

```
F-11 (Method F-9 (Backward-Chain F-1)) (BC-Meth F-10 F-1)
```

The queue now runs out since there are no other actions possible. At this point the ACCEPTOR is invoked. Seeing only one method available, the acceptor makes the obvious choice accepting the method proposed in F-11.

```
F-12 (Accepted F-11) (Acceptor F-11)
```

The acceptance of the method proposed in F-11 allows the method proposer to continue its work. This creates the following rule which represents the inference rule of Modus-Ponens:

```
R-11 (Rule ((:g (Implies Q P)) (*Rule* F-12 R-10)
            (:h Q))
      (Assert P (Modus-ponens :g :h)))
```

Notice that when the system created the rule R-11, it automatically generated a justification for it. New rules are created by the execution of a RULE expression which is typically nested within another rule; when this outer rule is triggered by some fact, the new rule is created with a justification indicating dependence on the outer rule and the triggering fact.

The body of the method proposer also creates the new sub-goal *q* triggering a series of assertions similar to those triggered by the original goal *p*. We get the following:

62 Explicit Control and The Task Network

```

F-15 (Satisfied F-10)          ()
F-17 (Refuted F-10)           ()
F-18 (Goal Q (P top-level))    (Sub-Goal F-12 (F-15 F-17))

R-3 (rule ((:f Q))             (Sat-goal F-10)
      (Assert (Satisfied F-10)
               (Satisfaction :f)))
R-4 (rule ((:f (not Q)))        (Ref-Goal F-10)
      (Assert (Refuted F-10)
               (Refutation :f)))
F-20 (Method F-10 (Backward-Chain F-2)) (BC-Method F-10 F-2)

```

The ACCEPTOR is now invoked and Method F-20 is accepted.

```

F-21 (Accepted F-20)           (Acceptor F-20)

```

which in turn triggers the rule for backward chaining, resulting in:

```

F-22 (Satisfied F-24)          ()
F-23 (Refuted F-24)           ()
F-24 (goal R (Q P top-level))  (bc-sub-goal (F-21)(F-22 F-23))
R-5 (Rule ((:g (Implies R Q))  ("Rule" F-21 R-0)
      (:h R))
      (Assert Q (Modus-ponens :g :h)))

```

At this point the necessary facts are available allowing rule R-5 to run on the fact F-24. Thus, we obtain:

```

F-25 Q (Modus-Ponens F-3 F-2)

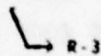
```

However, Q now triggers the rule R-3 which is watching for an assertion satisfying the goal F-18 (GOAL Q (P TOP-LEVEL)). This causes a justification to be added to the *satisfied* assertion, F-15, which was created when the goal F-18 was created:

F-15 (Satisfied F-16) (Satisfaction F-23)

However, the goal assertion F-18 depends on the *outness* of F-15 and the *method* and *show* assertions F-20 and F-21 depend on F-18. A quick inspection of the justifications will show that the following support structure exists at this time:

F-15 \leftrightarrow F-18 \rightarrow F-20 \rightarrow F-21 \rightarrow R-5



where a double headed arrow indicates non-monotonic support. Thus, when F-15 comes *in*, all of the other assertions go *out*. Notice, however, that all of these are control assertions. The fact assertion F-25 Q depends only on F-3 and F-2; it stays *in*. Furthermore, F-25 triggers the rule R-2 which represents the modus-ponens deduction for Q and (IMPLIES Q P). We obtain:

F-26 P (Modus-Ponens F-23 F-1)

As before, this triggers a *goal-satisfied* rule, this time R-1 for the goal F-9.

F-6 (Satisfied F-9) (Satisfaction F-24)

which causes the goal assertion F-9 to go *out*. A similar chain of dependencies to that above causes assertions F-11, F-12, and rules R-1, R-2 to go *out* as well. This leaves us with only the following useful assertions.

64 Explicit Control and The Task Network

F-1	(Implies Q P)	(Premise)
F-2	(Implies R Q)	(Premise)
F-3	R	(Premise)
F-4	(Subgoal P (top-level))	(Premise)
F-6	(Satisfied F-9)	(Satisfaction F-24)
F-15	(Satisfied F-18)	(Satisfaction F-23)
F-25	Q	(Modus-Ponens F-3 F-2)
F-26	P	(Modus-Ponens F-23 F-1)

Of course, this entire deduction might have been achieved more easily by a simple forward chaining rule for modus ponens. However, I have gone through this detail to illustrate the steps of the protocol. In general, blind forward chaining is a bad strategy since it allows uncontrolled deductions to lead into endless loops. Suppose that we added the following facts to the data-base:

```
(Implies (Number :x) (number (plus 1 :x)))
(Number 1)
```

Then the modus ponens rule would trigger infinitely often, filling the data-base with assertions of the form:

```
(number 1)
(number (plus 1 1))
(number (plus 1 (plus 1 1)))
...
```

Obviously, such infinite counting chains cannot be allowed to occur. On the other hand, it is desirable to allow some deductions to proceed in a forward manner. I have so far found it convenient to have both an antecedent and a consequent modus ponens rule; however, one must, therefore, avoid writing implications such as the one above. In the particular environment in which REASON operates one rarely wants to state implications like the one above anyhow. As we will see later, most of REASON's knowledge is expressed in the form of descriptions of data-structures using a specially designed specification language. This allows the knowledge acquisition portion of the system to build rules which correspond to these specifications and which do not engage in uncontrollable forward chaining.

It should be pointed out that one of the distinct advantages of the regimen of explicit control is that it is quite simple for the system to determine that it is engaging in infinite loops. If, for example, the same pattern appears as a subgoal of itself, then the system can decide not to pursue that subgoal by marking it as a loop and never accepting it. Similarly it can set itself limits on how deep into a case analysis it will go before deciding that it's on a losing course. Indeed, although our work has not yet progressed this far, the reasoning system can, in principle, reason about what that limit ought to be given the particular circumstances it which it currently finds itself. This begins to suggest the idea of the reasoning system having a "state of mind".

Section 4.1: Hypotheticals

The actual protocol is, in fact, more complicated than what I have illustrated so far. The complication is caused by the use of hypotheticals, particularly in conditional proofs. As a paradigmatic case consider the following problem (I will omit the refutation assertions in this example for the sake of brevity):

Given:	(Or A B)
	(Implies A C)
	(Implies B C)
	(Implies C D)
To Show:	D

REASON might attack this problem in a manner similar to that employed above, backward chaining from the goal *D* to both *A* and *C*. At this point it recognizes that case-splitting the disjunction (Or A B) is an appropriate method. This creates a set of conjunctive sub-goals, in this case, (Implies A D) and (Implies B D). Each of these is proven by the standard conditional proof method, assuming the antecedent and attempting to prove the consequent. The following is an excerpt of the assertions that result:

F-30	(or a b)	(promise)
F-31	(implies a c)	(promise)
F-32	(implies b c)	(promise)
F-33	(implies c d)	(promise)
F-54	(satisfied F-55)	(); No support, therefore I'm <i>OUT</i>
F-55	(goal d (top-level))	(subgoal (F-47) (F-54))
R-1	(Rule ((:f d))	(goal-sat F-55)
	(Assert '(satisfied F-55)	
	(Goal-Sat :f)))	
F-65	(method F-55 (splitting F-30))	(method F-55)
F-67	(accepted F-65)	(acceptor F-65)
F-99	(satisfied F-100)	; No support, therefore I'm <i>OUT</i>
F-100	(goal (implies a d) (d (top-level)))	(subgoal (F-74) (F-99))
R-2	(Rule ((:f (implies a d))	(goal-sat F-100)
	(Assert '(satisfied F-100)	
	(Goal-Sat :f)))	
F-105	(method F-100 (standard-implication))	(si-method F-100)
F-110	(accepted F-105)	(acceptor F-105)
F-112	a	(imp-assume (F-110) (F-113))
F-113	(Not a)	()
F-122	c	(mp F-31 F-112)
F-125	(satisfied F-126)	(); No support, therefore, I'm <i>OUT</i>
F-126	(goal d ((implies a d) d (top-level)))	(subgoal (F-110)(F-125))
R-3	(Rule ((:f d))	(goal-sat F-126)
	(Assert '(satisfied F-126)	
	(Goal-Sat :f)))	
F-127	d	(mp F-33 F-122)

Notice the rules R-1, R-2 and R-3 which correspond to the goals F-55, F-100 and F-126. Two of these (R-1 and R-3) are waiting for the same fact (F-127 d) to come in at which time each will rule will assert that its goal (F-55 and F-126 respectively) is satisfied. This, however, is an obvious mistake. If F-127 d comes in at this stage, this does not imply that d is true, only that d follows from the assumption a. Thus, the goal F-55 should not be satisfied by this occurrence of a.

The classic solution to this problem in AI systems such as **PLANNER** [Hewitt, 1972] **CONNIVER** [McDermott 1973], and **QA4** [Rulifson, et. al., 1972] is to use a context mechanism to represent the "echelon" in which the implication will be derived. Typically, a new context is created in which the assumption *A* as well as the new goal *D* (or its analogue) are asserted. When the fact *D* comes in, the satisfied assertion is added to the new context which is then discarded; only (IMPLIES *A D*) is returned to the old context. The problem with this approach is that it is altogether possible that the fact *D* derived in this new context might not depend on the assumption *A*, in which case the system ought to assert that the main goal *D* is satisfied and terminate the process; the context system is incapable of doing this since the contexts do not represent logical dependency but only the chronology of the problem solver's behavior. Lacking any representation of logical dependency, a context system becomes overly rigid.

REASON instead uses the dependencies maintained by the truth maintenance system as well as the explicit control assertions to guide itself to appropriate conclusions. Goal assertions contain within them a goal stack, indicating the chain of subgoals which led to the current goal. Although each such assertion includes a (linear) stack, the set of all such assertions is a (potentially non-linear) network, since the same goal may be reached by several different paths. The stack is included in goal assertions for two reasons: First it allows two nested occurrences of the same goal (such as *D* above) to be distinguished by their goal stacks. Since the two goals are represented by different assertions we may easily say that only one is satisfied while leaving the other to remain as an active goal.

The second reason for this representation is connected with the use of hypotheticals such as the assumption *A* made in trying to prove (Implies *A D*). The goal assertions actually used in **REASON** are an extension of those I have shown so far, including not only the goal stack, but also a set of assumptions made as part of the deduction process. These assumptions are referred to as the *context*, although this context should be distinguished from that of **CONNIVER** or **QA4**, in that the assumption set is an unordered set, while **CONNIVER** and **QA4**'s contexts are strictly nested. Every time a new assumption is made for the sake of hypothetical reasoning (as in proofs of implications or in indirect proofs) the newly assumed fact is added to the context part of the goal assertion associated with that assumption. To facilitate this, the primitive **GOAL-ASSERT** takes one more argument than shown above, namely the assumption context. Thus, the same set of assertions as shown above will now be represented as follows:

F-30	(or a b)	(premise)
F-31	(implies a c)	(premise)
F-32	(implies b c)	(premise)
F-33	(implies c d)	(premise)
F-54	(satisfied F-55)	(); No support, therefore I'm <i>OUT</i>
F-55	(goal d for (top-level) in ())	(subgoal (F-47) (F-54))
R-1	(Rule ((:f d)) (Assert '(satisfied F-55) '(Goal-Sat :f)))	(goal-set F-55)
F-65	(method F-55 (splitting F-30))	(method F-55)
F-67	(accepted F-65)	(acceptor F-65)
F-99	(satisfied F-100)	; No support, therefore I'm <i>OUT</i>
F-100	(goal (implies a d) for (d (top-level)) in ())	(subgoal (F-74) (F-99))
R-2	(Rule ((:f (implies a d)) (Assert '(satisfied F-100) '(Goal-Sat :f)))	(goal-set F-100)
F-105	(method F-100 (standard-implication))	(si-method F-100)
F-110	(accepted F-105)	(acceptor F-105)
F-112	a	(imp-assump (F-110) (F-113))
F-113	(Not a)	(); No support, therefore I'm <i>OUT</i>
F-122	c	(mp F-31 F-112)
F-125	(satisfied F-126)	(); No support, therefore, I'm <i>OUT</i>
F-126	(goal d for ((implies a d) d (top-level)) in (a))	(subgoal (F-110)(F-125))
R-3	(Rule ((:f d)) (Assert '(satisfied F-126) '(Goal-Sat :f)))	(goal-set F-126)
F-127	d	(mp F-33 F-122)

In *goal* assertions the keyword "for" indicates the subgoal stack while the keyword "in" indicates the assumption context. When the fact F-127 d comes *in* now, it is possible to determine which goal it actually satisfies. The rule is as follows:

For Complex Program Understanding

70 Explicit Control and The Task Network

Given an assertion P which triggers the pattern of a rule which is watching for the satisfaction of some goal

1. Request the Truth Maintenance System to prepare a list of all assumptions which support the satisfying fact P.
2. Fetch all goal statements whose goal matches the satisfying fact P.
3. For each goal assertion test whether the assumptions found in step 1 are a subset of the assumptions listed in the goal assertion's context list.
4. Discard those goal assertions which fail the test in 3.
5. For each of the remaining goal assertions in 4 assert that the fact P satisfies the goal assertion.

This procedure results in the following assertion when applied to the situation described above:

F-120 (satisfied F-126)

(goal-found f-127 f-112)

However, REASON does not assert that the original goal assertion F-55 is satisfied since it has an empty assumption context while the assertion F-127 depends on the assumption F-112 A.

This process is driven by the rules which watch for the presence of goal-satisfying facts, yet these rules in the current system are required to be stated in a more procedural manner than desired. The algorithm we have just stated is one which determines whether a certain pattern of dependencies obtains and acts only in that case. Given the philosophy of explicit control in a rule based system, it would be preferable to include such dependency patterns in the triggering list of a rule, simply allowing it to run whenever the appropriate combination of support and facts obtains. Unfortunately, the current mechanisms are considerably too weak to implement these desiderata and we are forced to employ more awkward mechanisms. What one would like is to be able to write something like the following:

```
(Rule ((:f1 (goal :g for :stack in :context))
      (:f2 :g)
      (? (subset (assumption-support :f2) :context)))
      (assert '(satisfied :f1) '(goal-found :f2)))
```

where the intention is to treat the third clause as if it were a fact, triggering the rule whenever the condition expressed in this clause is true. This, however, puts a burden on the truth maintenance system to not only be aware of changes in the *in* and *out* statuses of facts, but also to be aware of more complex conditions, signalling these to the reasoning system as well.

This poses an interesting research direction for future work which I have not yet pursued. Is it possible to develop a lexicon of such useful justification patterns and to extend the truth maintenance system to support the facility just outlined?

The partial solution I have adopted is to have a special kind of rule called a *trigger rule* which runs each time all its pattern's come *in*. The body of this rule is then free to perform further checks to determine if it wants to proceed; in particular, it can investigate the support patterns of various assertions. If these patterns of support are appropriate, the rule can then add assertions to the data-base. If not, it can simply exit; however, if the triggering patterns of the rule all become *in* at some later time, then the trigger rule will execute again, allowing the support pattern to be checked once more.

This treatment of trigger rules is different from normal rules. When a normal rule matches a set of facts, it is run on this set of facts exactly once: the first time that all the facts are *in*. Normal rules are concerned with truth; they implement deductions which are thereafter maintained by the truth maintenance system. When a rule runs, its job is to create new facts and to provide the TMS with a justification for each of these. As we've seen, a justification consists of two sets of facts: the *in* antecedents and the *out* antecedents. These later are used only in assumptions. The TMS views justifications as permanent implications: the *in*ness of the *in* antecedents together with the *out*ness of the *out* antecedents implies the *in*ness of the consequent.

Thus, a rule concerned with truth need only run once on any set of triggering facts since the pattern of support it creates is eternal and can be handled by TMS without further executions of the rule. Trigger rules are, in contrast, concerned with utility and control, concepts of much greater plasticity; they, therefore, require the additional flexibility of executing any time their trigger facts change status to *in*.

Once the trigger rule has executed, REASON concludes that the implication (IMPLIES A D) is proven. This is justified by a conditional proof justification; the computed support is independent of F-112 and includes only F-31 (IMPLIES A C) and F-33 (IMPLIES C D).

REASON moves on to the second half of the case-split. This proceeds in exactly the same manner as above. In this half, F-130 B is assumed which leads to F-122 C by Modus Ponens. The TMS already has a justification which says that if F-122 is *in*, then F-127 should be *in* as well; therefore, F-127 is brought *in* and the trigger rules run again. The trigger rule for the first side of the split is *out* since it depended on the goal F-55 which, since it is satisfied, is *out*; the trigger rule for the second side of the split and for the main goal F-55 are *in*. This time they conclude that the new sub-goal D is satisfied; therefore, (IMPLIES B D) is proved. Its conditional proof justification computes that the support includes only F-32 (IMPLIES B C) and F-33 (IMPLIES C D). The main goal F-55, however, still cannot be satisfied, since F-127 D still depends on F-130 A, an assumption not in F-55's context set.

However, the case analysis is now completed; F-127 D has been derived from both sides of the split. It is asserted with its justification pointing at the disjunction used for the case analysis and the implications proven on each side of the split. The trigger rule associated with F-55, the main goal, is still *in* and is triggered once again. This time it succeeds since the current support of F-127 includes no assumptions at all, but only the facts F-30, F-31, F-32, F-33.

Section 4.2: The Rules Of Inference

In the following pages, I will present the rules of inference currently used in the experimental system. These should not be regarded as finished products, but as experiments along the way. Similar rules may be found in [Doyle, 1977]. These rules are essentially the normal rules of standard logic embedded within the discipline of explicit representation of control information. It should be noted that these rules have the property that no non-control assertion will ever depend on a control assertion; thus, the logical validity of normal assertions is independent of the control regime.

The reader should also note that I do not show here those rules which are responsible for selecting among competing methods. These are still under development.

Contradiction Signalling

signal a contradiction to the TMS if both a fact and its negation are in

```
(Rule ((:f :p)
      (:f (not :p)))
      (assert '(and :p (not :p)) '(contradiction :f :p)))
```

Double Negation Simplification (Not Elimination)

```
(Rule ((:f (Not (not :p))))
      (assert :p '(double-negation-elim :f)))
```

74 Explicit Control and The Task Network

Antecedent Use of If-Then-Else (If-then-else elimination)

If the antecedent of an IF-THEN-ELSE is *in* assert the fact for the THEN branch. If the negation of the antecedent is *in*, assert the fact for the ELSE branch.

```
(rule ((:f (if :cond then :true else :false)))  
  (rule ((:g :cond))  
    (assert :true '(if-then-else-true :f :g)))  
  (rule ((:g (not :cond)))  
    (assert :false '(if-then-else-false :f :g))))
```

Modus Ponens (Implication Elimination)

```
(rule ((:f (implies :a :b))  
  (:g :a))  
  (assert :b '(mp :f :g))))
```

Quantifiers

The quantifiers used in REASON are slightly different than those of normal logic systems. The universal quantifier is a restricted quantifier with the force of an implication, for example:

```
(for-all (:x) (Member the-table :x)  
  (not (Key-part :x key-1)))
```

which states that no member of the table has key equal to key-1.

The existential quantifier is always coupled with a *such-that* clause which has the force of a conjunction, e.g.

```
(there-is (:x) (Member the-table :x)  
  such-that (key-part :x key-1))
```

Each quantified statement includes a list of variables bound by the quantifier; the first clause of either type of quantified statement must mention all variables in this list; the second clause may not have any free variables.

Antecedent Use of Universal Quantification (Universal Elimination).

If an object exists which matches the antecedent of the FOR-ALL, then assert the consequent clause (with the bindings substituted in).

```
(rule ((:f (for-all (:vars :p :q))
      (:g :p))
      (assert :q '(:for-all :f :g))))
```

For example, if

```
(For-all (:x) (member list-1 :x)
  (there-is (:y) (sublist list-1 :y)
    such-that (first :y :x)))
```

is matched against

```
(Member list-1 Object-22)
```

we would then eliminate from the universal, obtaining:

```
(There-is (:y) (sublist list-1 :y)
  such-that (first :y Object-22))
```

Universal Introduction

```

(rule ((:f (goal (for-all :vars :p :q) for :goal in :context)))
  (propose-method
    '(Method :f (typical-member)) '(UITM :f)
    (let ((subst (ui-build-subs :vars)))
      (let ((:newp (instance :p subst))
            (:newq (instance :q subst)))
        (goal-assert '(implies :newp :newq)
                      '(((for-all :vars :p :q) . :goal)
                        :context
                        '(for-all-sub-goal :f))))
      (rule ((:g (implies :newp :newq)))
        (assert '(for-all :vars :p :q) '(ui :g))))))

```

Where `UI-BUILD-SUBS` binds each variable in `:VARS` to a newly created anonymous object (see next section). An anonymous object is one whose identity is unknown; it is *a priori* impossible to tell whether an anonymous object is identical to another object. `UI-BUILD-SUBS` also marks each anonymous object it creates as a `UI-OBJECT` (using the object's property list); this mark is used by the existential introduction and elimination rules to prevent certain logical bugs explained below.

Expansion of Existential Quantification on Demand (Existential Elimination)

The reasoning system occasionally will want to expand an existential, replacing the variable of the quantified statement by an *anonymous object*. However, if the existential contains *UI-OBJECTS* (objects created for Universal Introduction) then the anonymous objects created are marked to show their dependency on the *UI-OBJECTS*.

```
(rule ((:f (there-exists :vars :fact such-that :pred)))
  (rule ((:g (expand :f)))
    (let ((:subst (ee-build-subs :vars :pred :fact)))
      (let ((:new-fact (instance :fact :subst))
            (:new-pred (instance :pred :subst)))
        (assert :new-fact '(there-is :f))
        (assert :new-pred '(there-is :f))))))
```

where *EE-BUILD-SUBS* creates a substitution which binds each of the variables in *:VARS* to a newly created anonymous object. *INSTANCE* is a function which substitutes these bindings into its second argument. *Expand* assertions such as referred to in the above rule are asserted occasionally during symbolic program evaluation.

As mentioned *EE-BUILD-SUBS* marks the newly created anonymous objects for their dependence on *UI-OBJECTS*. For example, if *OBJECT-1* is a *UI-OBJECT* and we have the following:

```
(There-is (:y) (Member object-1 :y) such-that (key :y key-1))
```

and existential elimination creates *MEMBER-1*, an anonymous object, and substitutes this for *:y*, then *EE-BUILD-SUBS* will mark *MEMBER-1* to show its dependence on *OBJECT-1*. We will obtain the assertions

```
(Member object-1 member-1)
(Key member-1 key-1)
```

and the property list of *MEMBER-1* will be marked so that the *UI-DEPENDENCY* property of *MEMBER-1* is the list (*OBJECT-1*).

Existential Introduction

```

(rule ((:f (goal (there-is :vars :p such-that :q) for :goal in :context)))
  (Propose-Method
    '(Method :f (Standard-EI)) '(EISM :f)
    (goal-assert :p
      '((there-is :vars :p such-that :q) . :goal)
      :context
      '(there-is-sub-goal-1 :f))
    (rule ((:g :p))
      (cond ((ui-object-free :p :vars)
        (goal-assert :q
          '(:p . :goal)
          :context
          '(there-is-sub-goal-2 :f :g))
        (rule ((:h :q))
          (Cond ((ui-object-free :q :vars)
            (assert '(there-is :vars :p such-that :q)
              '(exist-intro :h :g))))))))))

```

The function `UI-OBJECT-FREE` checks to see whether the matching has resulted in a variable of either `:p` or `:q` becoming bound to an expression containing an anonymous object which is marked with a `UI-DEPENDENCY` property. If so, this expression cannot be used to introduce an existential. This prevents the classic mistake of using

```
{for-all (:x) (object :x) (there-is (:y) (object :y) such-that (P :x :y))}
```

to conclude:

```
(There-is (:y) (object :y) such-that (for-all (:x) (object :x) (P :x :y)))
```

(where `object` is a predicate true of everything, and is used merely to fill the first position of my restricted quantifier notation).

There are some delicate issues of control involved in simultaneous sub-goals which share variables. These come up in proving existential quantifiers like the ones used here. [Doyle, 1977] discusses these problems and presents a more advanced solution than I have used here.

Expansion of And (And Elimination)

```
(rule (:f (and :p :q))
  (assert :p '(and-elim :f))
  (assert :q '(and-elim :f)))
```

Disjunctive Elimination (Or Elimination)

This is done in two parts. When a disjunct is asserted, it is put into an expanded form so that rules can easily determine whether a particular fact is a clause of the disjunction. Then, if a clause in a disjunction is ever negated, a new disjunction can be asserted, including all the other clauses of the old disjunction.

Set Up Expanded version of Disjunct

```
(rule ((:f (or . :x)))
  (do ((dis :x (cdr dis)))
    ((null dis))
    (let ((:dis (disjunct-of :f ,(car dis))))
      (assert :dis '(spread-disjuncts :f))))))
```

The double quote (") is a macro which produces a list. The items inside the list are not evaluated unless preceded by a comma (,). Variables (e.g. :f) are always evaluated. LET contains two parts: a set of binding expressions and a body. Each binding expression contains a variable and an expression. The variable is bound to the value of the expression. The body is executed in the environment created by these bindings.

Do The Actual Work When Appropriate

```
(rule ((:f (disjunct-of :d :p))
  (:g (not :p)))
  (assert '(or . ,(safe-delete :p :d)) '(disjunctive-elim :f :g)))
```

For Complex Program Understanding

80 Explicit Control and The Task Network

The exclamation point character (!) is a macro which when applied to a fact name produces a fact statement. I.e. `!d = (Or ...)` in the above. `SAFE-DELETE` is a variant of the built-in LISP function `DELETE` which does not side-effect its argument.

Simplify singleton or

If the above leads to a disjunction with only one clause, that clause may be asserted.

```
(rule ((:f (or :p)))  
      (assert :p '(or-simplification :f)))
```

The Basic conjunctive goal mechanism:

This is essentially a sub-routine called by other strategies. The routine will try each of the sub-goals in turn. A later sub-goal will not be tried until the prior sub-goal is satisfied. A refutation will stop the iteration, asserting that the calling goal is refuted. Thus, this should be called only when the conjunction of the sub-goals is equivalent to the calling goal.

The call is made by asserting:

```
(conjunctive-goals :first :rest :dep :stack :context)
```

where the arguments have the following meaning:

1. `first`: the first sub-goal to attempt.
2. `rest`: the remaining sub-goals to attempt after the first is satisfied.
3. `dep`: previously accumulated assertions upon which the final goal will depend. Each time a sub-goal is satisfied, the satisfying fact is added to this argument.
4. `stack`: the goal stack with which the invocation was made. The first item on this stack is the immediately dominating goal. If all the sub-goals are satisfied, we can conclude that the first element on the stack is satisfied.
5. `context`: the assumption context in which this is invoked.

Conjunctive Sub-Goal Mechanism

```

(rule ((:f (conjunctive-goals :first :rest :dep
      (:top . :stack) :context)))
  (goal-assert :first '(:top . :stack) :context '(conjunctive-goals :f))

(rule ((:g (not :first)))
  (Assert '(Not :top) '(Conj-goal-refutation :g)))

(rule ((:g :first))
  (cond
    (:rest
     (assert
      '(conjunctive-goals (car :rest) (cdr :rest) (:g . :dep)
        (:top . :stack) :context)
      '(conjunctive-goal-control :f :h)))
    (t
     (assert :top '(conjunctive-goals :g . :dep))))))

```

Proof by cases

Split a disjunction, creating a case analysis. In each case attempt to show that the current clause of the disjunction implies the desired goal. This is done by creating a sub-goal which is the conjunction of these implications. Conjunction Introduction will invoke the conjunctive goal mechanism to conduct the proof.

```

(rule ((:g (goal :p for :goal in :context))
  (:h (or . :q)))
  (Propose-Method '(Method :g (splitting :h)) '(Split-Meth :g :h)
    (let ((:qf '(and . . (mapcar '(lambda (x) '(implies .x .:p)) :q)))
      (goal-assert :qf '(:p . :goal) :context
        '(use-conj-goals-for-case-split :g :h)))))

```

Implication Introduction

```

(rule ((:f (goal (implies :a :b) for :goal in :context)))
  (Propose-Method '(Method :f (Standard-imp-intro)) '(SII :f)
    (goal-assert :b
      '(((implies :a :b) . :goal)
        '(:a . :context)
        '(implies-subgoal :f))
    (assume :a '(standard-implies-rules-assumption :f))
    (rule ((:h :a)
      (:g :b))
      (assert '(implies :a :b) '(cp :g (:h)))))))

```

Disjunction Introduction

```

(Rule ((:f (goal (Or . :d) for :goal in :context)))
  (Propose-Method '(Method :f (standard-dis-intro)) '(SDI :f)
    (do ((dis :d (cdr dis)))
      ((Null dis))
      (let ((:car-dis (car dis)))
        (goal-assert :dis '(((Or . :d) . :goal) :context '(disj-intro :f))
          (Rule ((:g :car-dis))
            (Assert '(Or . :d) '(dis-intro :g)))))))

```

Conjunction Introduction

```

(Rule ((:f (goal (And :cf . :cr) for :goal in :context)))
  (Propose-Method '(Method :f (standard-conj-intro)) '(SCI :f)
    (Assert
      '({conjunctive-goals :cf :cr () ((And :cf . :cr) . :goal) :context)
      '(Use-conj-goals-for-and-intro :f))))

```

If-Then-Else Introduction

```
(rule ((:f (goal (if :p then :q else :r) for :goal in :context)))
  (Propose-Method '(Method :f (standard i-t-e)) '(SITEI :f)
    (Assert
      '(conjunctive-goals (implies :p :q) ((implies (not :p) :r)) ()
        ((if :p then :q else :r) . :goal) :context)
      '(use-conj-goals-for-i-t-e :f))))
```

Backward Chaining

```
(rule ((:f (goal :q for :goal in :context))
  (:g (implies :p :q)))
  (Propose-Method '(Method :f (backward-chaining :g)) '(BC :f :g)
    (goal-assert :p '(:q . :goal) :context '(backward-chain :f :g))))
```

Backward Chaining Through If-then-Else

```
(Rule ((:f (goal :q for :goal in :context))
  (:g (if :p then :q else :r)))
  (Propose-Method '(Method :f (if-then-back-chain :g)) '(ITEBC :f :g)
    (goal-assert :p '(:q . :goal) :context '(i-t-e-back-chain :f :g)))

(Rule ((:f (goal :r for :goal in :context))
  (:g (if :p then :q else :r)))
  (Propose-Method '(Method :f (if-else-back-chain :g)) '(ITEBC :f :g)
    (goal-assert '(not :p) '(:r . :goal) :context '(i-t-e-back-chain :f :g))))
```

Indirect Proof

```
(Rule ((:f (goal :p for :goal in :context))
  (Propose-Method '(Method :f (indirect-proof)) '(IP :f)
    (Assume '(Not :p) '(indirect-proof-assumption :f))))
```

For Complex Program Understanding

84 Explicit Control and The Task Network

If this leads to a contradiction and if this contradiction actually depends on the assumption of (NOT :P), then the truth maintenance system will determine that the assumption is responsible for the contradiction. It will then bring *p* *in*. This will *out* the assumption. This process is called dependency-directed backtracking; it is described in [Doyle, 1978] and in [Stallman & Sussman, 1977].

Contrapositive Deduction

```
(Rule ((:f (Implies :p :q))
      (:g (Not :q)))
      (Assert '(not :p) '(contrapositive :g :f)))

(Rule ((:f (goal (Not :p) for :goal in :context))
      (:g (Implies :p :q)))
      (Propose-Method '(Method :f (Contrapositive-chaining :g)) '(CPC :f :g)
        (goal-assert '(not :q) '((Not :p) . :goal) :context
          '(contrapositive-back-chain :f :g))))
```

DeMorgan's Rules

```
(Not (and a b c ...)) => (or (not a) (not b) (not c) ...)
(Not (or a b c ...)) => (And (not a) (not b) (not c) ...)

(rule ((:f (not (and . :x))))
      (let ((:disj '(or . ,(mapcar '(lambda (x) '(not ,x)) :x))))
        (assert :disj '(de-demorgan :f))))

(rule ((:f (not (or . :x))))
      (let ((:conj '(and . ,(mapcar '(lambda (x) '(not ,x)) :x))))
        (assert :conj '(demorgan-demorgan :f))))
```

Negation and Quantifiers

```

(Not (For-all :vars :p :q)) <=> (There-is :vars :p such-that (not :q))
(Not (There-is :vars :p such-that :q)) <=> (For-all :vars :p (not :q))

(Rule ((:f (not (For-all :vars :p :q))))
  (Assert '(There-is :vars :p such-that (not :q))
    '(negated-for-all :f)))

(Rule ((:f (not (there-is :vars :p such-that :q))))
  (Assert '(For-all :vars :p (not :q))
    '(negated-there-is :f)))

(Rule ((:f (goal (not (For-all :vars :p :q)) for :goal in :context)))
  (Propose-Method '(Method :f (standard)) '(Quant :f)
    (goal-assert '(there-is :vars :p such-that (not :q))
      '((not (for-all :vars :p :q)) . :goal)
      :context
      '(negated-for-all-standard-sub-goal :f))))

(Rule ((:f (goal (not (there-is :vars :p such-that :q)) for :goal in :context)))
  (Propose-Method '(Method :f (standard)) '(Quant :f)
    (goal-assert '(for-all :vars :p (not :q))
      '((not (there-is :vars :p :q)) . :goal)
      :context
      '(negated-there-is-standard-sub-goal :f))))

```

Section 4.3: Closing The Reflexive Loop

So far I have shown the use of the task agenda only in the context of theorem proving. However, the protocol shown above is actually the way REASON does anything which needs to be open to introspective control. Tasks other than theorem proving goals are entered into the system by making a TASK assertion which is treated in much the same manner as the GOAL assertions. Such assertions stimulate the proposal of methods; methods are chosen by the ACCEPTOR just as was shown above.

Frequently some partial ordering must be imposed on the execution of tasks. This is done by asserting a CONTROL-FLOW assertion mentioning the two tasks which are to be ordered. Similarly a task can make information available for use by asserting that some object is one of its outputs. This information can be propagated to other tasks by the assertion of DATA-FLOW assertions which mention the output port of the first task

For Complex Program Understanding

and an input port of some other task.

Of course this is just the primitives which are used to describe programs in the plan diagram formalism!! In fact, I am currently working on building a catalogue of useful plans for use by REASON itself. These will be coupled with a set of rules which state that a useful way to accomplish some task is to apply one of the plans from the catalogue. Of course these plans create sub-tasks and REASON will have to choose methods to accomplish each of these. However, it is often the case that there is an *a priori* good choice for some of the sub-tasks of a particular plan. Thus, extremely useful pragmatic information can be provided to the ACCEPTOR by building rules which analyze the history of method and plan selection and use this analysis to select methods. This is roughly the approach followed in [McDermott, 1977].

Although this is so far past the current development of REASON that it now seems to be science fiction, there is yet another advantage to this approach. REASON is itself a program written to analyze programs; the language in which a substantial part of REASON is written is the language it is capable of analyzing. Therefore, it is possible for REASON to analyze itself!! (At least in principle, it is possible).

In later chapters, I will often refer to REASON'S protocols. These are sets of tasks to be entered into the task agenda; they are represented in the plan language. In the next section I will develop some more representations used in the reasoning system; I will then turn to a deeper look at the plan language.

Section 4.4: Equality, Reference and Anonymous Objects

If one wishes to show that a program has a certain property one must show that whatever inputs the program is given, it will still behave in accordance with the property. The method used in REASON is to evaluate the behavior of the program when presented with typical inputs. *Anonymous Objects* [Hewitt, 1975], [Rich & Shrobe, 1976], [Moore 1975], (they are called *formal objects* in [Sussman, 1975]) are used to represent such typical inputs. An anonymous object is one whose identity is unknown in the sense that given an anonymous object and any other object, it is *a priori* unknown whether or not the two objects are identical (in the sense of being the same object).

AD-A078 055

MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 6/4
DEPENDENCY DIRECTED REASONING FOR COMPLEX PROGRAM UNDERSTANDING--ETC(U)
APR 79 H E SHROBE
AI-TR-503

N00014-75-C-0643

NL

UNCLASSIFIED

2 OF 4
AD-
A078055



Anonymous objects provide a convenient stand-in for unknown information of various kinds. For example, suppose we wanted to say that the third field of the record input to PROCEDURE-1 is a sorted list. If we did not know exactly what item was input to PROCEDURE-1 we would have to make up some token to stand for that item and similarly for the third field of this record. We might do this as follows:

```
(input the-record procedure-1 Anon-1)
(third-field Anon-1 Anon-2)
(sorted-list Anon-2)
```

Notice, however, that the first two of these predicates are functional, they uniquely determine their last argument. It is, therefore, possible to use a more concise notation:

```
(sorted-list [third-field [input the-record procedure-1]])
```

Each expression in brackets ([...]) is read "the object which satisfies ..." and refers to the unique object which could appear in the final position of the equivalent predicate. In fact, such reference expressions can be constructed for predicates which are known to be functional in any position. For example, if we knew that there were a unique list which contains ENTRY-1 as a member, we could refer to this list as follows:

```
[Member :list Entry-1]
```

in which the variable :list indicates the position of the statement which is being referred to. The reference expressions above are further abbreviations of this notation in which the last position is the variable and is by convention simply dropped.

Whenever REASON asserts a statement with reference expressions in it, it attempts to *resolve* the references. Reference resolution involves two stages: First, if there is an object which satisfies the reference expression, that object is substituted for the reference expression. Second, if no such object exists, an anonymous object is created to satisfy the reference expression. For example, suppose the following is asserted:

88 Explicit Control and The Task Network

F-0 (Left pair-1 [right pair-2])

To process this assertion, REASON must resolve the reference expression

[right pair-2]

i.e. it must find an assertion matching the pattern

(right pair-2 :obj)

There are two cases. Suppose the data-base already contained the assertion:

F-1 (Right pair-2 The-Answer)

Then the processing would be completed by asserting:

F-2 (Left pair-1 The-answer) (Reference-Resolution F-0 F-1)

If, however, the data-base contained no assertion matching the pattern

(Right Pair-2 :Obj)

then the system would create the anonymous object OBJECT-1, and assert that it satisfies the reference expression. Notice that this assertion is not an assumption since it is not really saying anything new. The reference expression itself only says that there is some object satisfying the expression; resolving the reference by creating the anonymous object merely gives this object a name. Since the anonymous object created to resolve the reference might be equal to any other object, this new assertion cannot be false; therefore, it is justified as a premise, i.e. its truth does not depend on the truth of any other assertion.

Since resolving the reference expression creates a new assertion, processing may now proceed as above.

F-2 (Right pair-2 OBJECT-1) (Reference-resolver)
 F-3 (Left pair-1 OBJECT-1) (Reference-Resolution F-0 F-2)

Of course, there is no restriction on the nesting of reference expressions, so the processing is recursive.

Identification

The use of reference expressions raises the possibility that we might wind up with two distinct names for the same object. For example, in resolving the reference expression above we created the anonymous object OBJECT-1 to stand for the right part of PAIR-2 and from this we deduced that the left part of PAIR-1 is also OBJECT-1. However, suppose that we had the following assertion in the data-base:

F-4 (Left pair-1 Left-55) (some-justification ...)

Since a pair can only have one left part, it must be the case that LEFT-55 and OBJECT-1 are the same object. We are then faced with the problem of what to do with these two names for the same thing, i.e. the problem of handling equality. REASON uses a rather unusual tactic in this situation. The standard tactic in most reasoning systems is to build up equivalence classes of equal objects. This however, imposes a price when searching for a match since one must check for variants of the desired assertion using any possible representative of the equivalence class. REASON instead, eliminates this possibility by doing the work in advance; it makes one of the objects "disappear". In the current example, there is really no further use for the name OBJECT-1, since it was merely created as a stand-in when we lacked the information to know what the right part of PAIR-2 was. However, we have now deduced that information, so the stand-in is unnecessary.

Since we have learned that OBJECT-1 is really LEFT-55, we can substitute LEFT-55 for OBJECT-1 in any assertion in which OBJECT-1 occurs. This process is called *identification*. To make this possible, REASON maintains an index of which assertions the various objects occur in; the index is represented by assertions of the form

(Occurs-in <object> <fact-name>)

For Complex Program Understanding

90 Explicit Control and The Task Network

Needless to say, these assertions are not indexed with more *occurs-in* assertions. Every time a new assertion is added to the data-base, this index is updated. When an identification is required, it is then simple to retrieve the appropriate assertions and make the substitutions.

However, this process as explained so far would result in another problem. Namely, every assertion mentioning *object-1* is now paralleled by an equivalent assertion mentioning *left-55*. It would be wasted effort for both of these assertions to be retrieved every time some information was desired. *REASON*, therefore, maintains a mark on each assertion, called the *utility* mark which serves a function similar to that of the *in-out* mark. If an assertion has its utility mark set, then it is regarded as being useless, no rule will trigger on it and it will not be retrieved by a normal fetch request. However, it is still regarded as being true.

As *REASON* goes through the identification process, it sets the utility mark of each assertion which mentions the anonymous object being identified away (*object-1* in this case). Each new assertion depends on both the *io* assertion and the original assertion from which it was built. In our current example, we have:

F-2 (Right pair-2 *object-1*)
F-3 (Left pair-1 *object-1*)
F-4 (Left pair-1 *left-55*)

As noted, F-3 and F-4 imply that *object-1* and *left-55* are identical. Thus, an *io* assertion is derived, initiating the identification process. The following assertions result:

F-5 (ID *object-1 left-55*) (Part-identification F-3 F-4)
F-6 (Right pair-2 *left-55*) (Identification F-2 F-5)

In addition, both F-2 and F-3 will have their utility mark set. The following rules implement this process:

```

(trigger-rule ((:f (id :obj-1 :obj-2))
  (:g (occurs-in :obj-1 :fact-1)))
  (set-utility-mark :fact-1 'useless) ; mark the fact useless
  (let ((:new-fact (subst :obj-2 :obj-1 :fact-1)))
    (assert :new-fact (identification :f :g))))

(rule ((:f (part :obj-type :part-name))
  (:g (type :obj-type :obj)))
  (rule ((:h (:part-name :obj :part-1))
    (:i (:part-name :obj :part-2)))
    (or (eq :part-1 :part-2)
      (assert (id :part-1 :part-2)
        (part-identification :f :g :h :i))))))

```

It is important to understand the distinction made here between the utility mark and the notion of *in* and *out*. *In* and *out* deal with belief (or logical relationships) while the utility mark is strictly an issue of control (of heuristic value). A fact which is *in* but whose utility mark is set is still regarded as true (or believed) even though it will be ignored. This is crucial since the justification for fact *r-8* above is *r-2*, a fact whose utility mark is set. If *r-2* were regarded as not being believed (as opposed to simply not being useful) then *r-8* would have no support and would itself be *out*. A fact whose utility mark is set may support belief in other facts although its presence will otherwise be ignored.

Whenever the truth maintenance system notifies REASON that the belief status of an *is* assertion has changed from *in* to *out*, REASON will remove the utility mark from each assertion which had previously been marked. This situation arises frequently in hypothetical reasoning, when for sake of argument the system assumes that two objects are identical, leading to an identification process. Later when the system retracts this assumption, the *is* assertion will become *out* removing the support for the covered assertions. When the utility mark of an assertion is removed, the system will run any rule which matches the assertion but which has not yet executed.

Section 4.5: Situational Logic

So far I have ignored the need to represent the temporal behavior of programs. The temporal nature of a fact is indicated by tagging an assertion with a *situation* tag [McCarthy, 1968] indicating when that assertion is believed to be true; different points in time are represented by different situation tags. Thus, we might write:

```
((First list-1 obj-1) situation-1)
((First list-1 obj-2) situation-2)
```

to indicate that the first object in LIST-1 is OBJ-1 at one point of time while it is OBJ-2 at another. This does not imply that OBJ-1 is identical to OBJ-2; the actual rule for identification used in REASON requires that the situations of the two assertions be identical.

It is often the case that we need to make reference expressions within this temporal notation. This is indicated with a *temporal reference expression* which is denoted using braces ({ ... }). For example:

```
((First list-1 ((first list-2) situation-2) ) situation-1)
```

says that the object which is the first element of LIST-1 in SITUATION-1 is also the first element of LIST-2 in SITUATION-2. Notice that it does not follow from this that there is any situation in which the first element of LIST-1 is ever the same object as the first element of LIST-2. A temporal reference expression has two parts: the assertion expression and the situation expression; either of these may be a simple reference expression. Temporal reference expressions are handled in essentially the same way as simple reference expressions discussed above.

Some assertions are *trans-situational* in that they relate assertions or objects from different situations. 10 assertions are an obvious example of this. If two objects are identical, then they are identical in all situations (for all time as it were). Thus, 10 assertions are never situationally tagged.

Logical connectives may also be used to build trans-situational assertions. For example:

```
(Or ((first list-1 obj-1) s1) ((Rest list-1 obj-2) s2))
```

is trans-situational. It is not true in any situation, but rather is an assertion relating facts in different situations. These will be important in chapter 11 where I discuss reasoning about side effects.

REASON also allows assertions to relate the states of objects at different points of time. Suppose we wished to describe the behavior of the MACLISP `NREVERSE` program which reverses a list by changing the pointers in its cells. This program works by side-effect; we want to say that the list which is the output of this program is the reverse of the list which is the input. However, we are talking about the same list in both cases since the program works by side-effect. The output cell is the one which was the last cell of the input list; after the `NREVERSE` program has run, this cell is the head of the reversed list. Suppose that `s1` is the situation just before the `NREVERSE` program executes and `s2` is the situation just afterwards. We can then write:

```
(Reverse <[list-1 s1]> <[list-2 s2]>)
```

The expressions within angle brackets (`<[...]>`) is called an *object-state* expression. It is read as "the state of ... in situation ...". If an assertion mentions only object-state expressions and the situation part of each such expression is the same situation, then the assertion is equivalent to a situationally tagged assertion mentioning the objects of each object-state expression and tagged with the common situation tag.

```
(P <[x1 s1]> ... <[xi s1]> ... <[xn s1]>)<br>
=<br>
((P x1 ... xi ... xn) s1)
```

Notice that the `REVERSE` assertion above is not reducible to a simple situationally tagged assertion. However, consider what would happen if the `REVERSE` assertion were replaced by its definition. We define `REVERSE` recursively, saying that one list is the reverse of the other if the first object of one is the last object of the other and if the rest of the first list is the reverse of the fragment of the second list beginning with the first object and continuing up to the last. (I will be more specific about how such

For Complex Program Understanding

definitions are stated in chapter 10). We would then obtain the following:

```
(Reverse <[11 s1]> <[12 s2]>)  
<=> (And (First <[11 s1]> [last <[12 s2]> ] )  
        (Reverse [rest <[11 s1]>] [leading-fragment <[12 s2]> ]))
```

this will lead to reference resolution as shown earlier. However, as we begin to resolve the references we will see that many of the expressions are simple reference expressions (i.e. without situational tags) which involve only object state descriptions from a single state. For example, from the first clause we obtain the following:

```
[last <[12 s2]> ]
```

Using the rule stated above this is resolved to:

```
((last 12 anon-1) s2)
```

and the value of the reference expression is ANON-1. Thus, following the same rule, the enclosing clause becomes:

```
((first 11 anon-1) s1)
```

If we continue this process we will ultimately wind up with only simple situationally tagged assertions. Actually, we also wind up with a second trans-situational REVERSE expression. However, by induction, this will also lead to a set of simple situationally tagged assertions, none of which are trans-situational. If a defined relation including object-state descriptions can be reduced to simple assertions which are not trans-situational, then the relation makes sense as a trans-situational assertion. REVERSE is such a relation. Membership in a data-structure is not; an examination of the definition of LIST-MEMBERSHIP for example, shows that any trans-situational use of membership is incoherent.

A similar reduction can be applied to trans-situational assertions built from logical connectives in which each clause is tagged with the same situation.

(Or (P1 s1) ... (Pi s1) ... (Pn s1))

<=>

((Or P1 ... Pi ... Pn) s1)

A similar rule applies to negation:

(not (P s1)) <=> ((not P) s1)

Chapter 5: Describing Programs

I will use two distinct methods of describing program segments. The first of these, called *specs*, is a formalism for specifying a segment's input/output behavior. The second, called *plan-diagrams* is used to build a complex segment by connecting together simpler ones. Intuitively, the specs represent the properties of the program to be proved and the plan diagram represents the program. Analysis consists of showing that the behavior which results from a plan diagram is that required by the segment's specs.

Section 5.1: Specs - I/O Descriptions

Simple specs consist of 4 sets of clauses: Inputs, Outputs, Expects and Asserts. The first two of these are simply lists of internal names or *ports* for the data objects which are the inputs (outputs) of the segment being specified. The *expect* clauses are a set of requirements which must be satisfied at the time the segment is applied to its inputs. Typically these are type constraints or simple relationships between the input objects. Finally, the *assert* clauses are a set of conditions which are promised to hold immediately after the segment has finished its execution. The assert clauses may mention both input and output objects, providing a convenient method for describing side-effects on the input objects.

We can use the specs formalism to specify a program which calculates the fringe of a tree as follows:

```
(defspecs fringe
  (Inputs: the-tree)
  (Expect: (Object-type the-tree Binary-tree))
  (Outputs: the-fringe)
  (Assert: (Object-type the-fringe List)
    (For-all (:the-node) (leaf-node the-tree :the-node)
      (member the-fringe :the-node))
    (For-all (:the-node) (member the-fringe :the-node)
      (leaf-node the-tree :the-node))))
```

Spec clauses are written in a variant of the predicate calculus which uses the pattern matching syntax of artificial intelligence languages and which uses the situation tag notation of the situational calculus [McCarthy, 1968]. The identifiers preceded by colons (e.g. :THE-NODE) are variables; thus, the two quantifiers say that every leaf node of the tree is represented in the output of FRINGE and conversely that only the leaf nodes are represented. Where it is possible to unambiguously omit the situation tag on a predicate we do so. This is almost always possible, since the use of distinct input and output names for the same object defines which situation is meant. For example, the first clause of the first quantifier above only mentions input objects and is, therefore, taken to apply in the input situation. The second clause refers to an output object and, therefore, refers to the output situation. In cases where this is not possible the two special symbols *BEFORE* and *AFTER* are available as names of the input and output situations. When specs are used in the symbolic evaluation process, the symbolic evaluator defaults in the appropriate situational tags.

Specs may also have a case structure which reflects the ability of the segment to cause control branching. This is done by adding case clauses. For example a test which checks whether a node is a leaf node can be specified as follows:

```
(defspecs leaf?
  (inputs: the-node)
  (expect: (Object-type the-node binary-tree-node))
  (case-1:
    (when: (Object-type the-node Leaf)))
  (case-2
    (when: (not (Object-type the-node Leaf)))))
```

This says that when the input node is a leaf we take one control branch and when it's not we take the second branch. As above, the segment has expect clauses which must be satisfied. This segment produces no outputs and has no side-effects. There are, therefore, no outputs or assert clauses. Segments have any number of cases and these cases may have outputs and assert clauses nested within them. This allows us to specify complicated segments which create control branches as well as producing outputs. A LOOKUP routine for a complex data-structure might have such specs.

It is often necessary to state that several segments share the same I/O behavior. One reason for this is that there are tasks for which several distinct algorithms exist; these different algorithms lead to distinct segments, but their specs are identical. There is a second need for saying that different segments have identical I/O behavior. Consider the following code for the fringe program:

```
(defun fringe (tree)
  (fringe-1 tree nil))

(defun fringe-1 (tree acc)
  (cond ((leaf? tree)(cons tree acc))
        (t (fringe-1 (left tree)(fringe-1 (right tree))))))
```

Notice that there are two recursive calls to `FRINGE-1`. Thus, there are three instances of `FRINGE` which we might want to distinguish for some purposes while still maintaining the awareness that these segments have a common I/O specification.

The name in a `defspects` statement is therefore regarded as a *spec-type*, rather than as the specs for any particular segment. If we need to indicate that a segment has the specs in a `defspects` statement, we state that its spec-type is the spec-type-name in the `defspects` clause. Thus, we could say that the two recursive calls in the fringe program have the same I/O behavior as follows:

```
(spec-type left-fringe fringe)
(spec-type right-fringe fringe)
```

where `LEFT-FRIDGE` is the recursive call to `FRINGE` for the left sub-tree and `RIGHT-FRIDGE` is the recursive call for the right sub-tree. This does not yet allow us to state that these two instances of `FRINGE` have identical internal structure. That is the subject of the next section.

Section 5.2: Plan Diagrams

Plan diagrams are a method of building a program segment by linking together the behaviors of smaller segments. In talking about a plan diagram there will always be a main segment (i.e. the segment described by the plan diagram) and a set of sub-segments which are being linked to form the main segment. In turn, some of these sub-segments will have plan diagrams and internal segments of their own. Thus, there may be several levels of aggregation within a plan diagram.

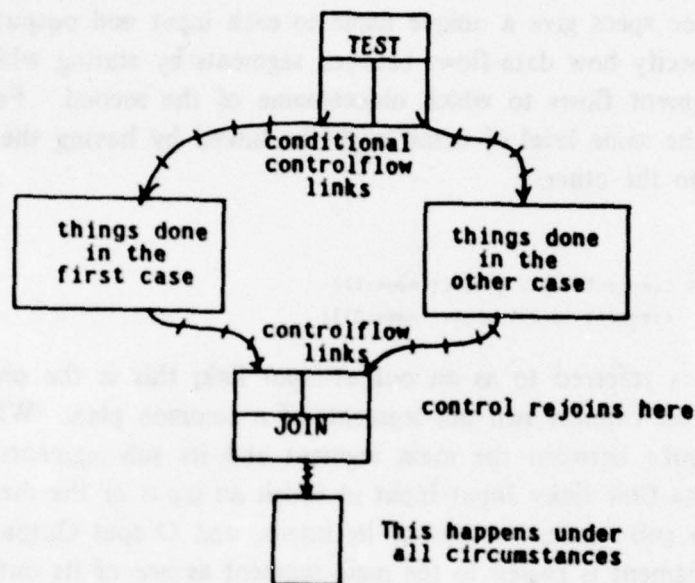
Segments within a plan diagram may be joined by two kinds of links: *data-flow* and *control-flow*. Since specs give a unique name to each input and output object of a segment, we may specify how data-flows between segments by stating which object-name of the first segment flows to which object-name of the second. For example two sub-segments at the same level of detail might be linked by having the output of one flow as an input to the other.

```
(dataflow (output <segment-id-1> <object-name-1>)
          (input  <segment-id-2> <object-name-2>))
```

The data-flow above is referred to as an output-input link; this is the only type of data-flow link which can connect two sub-segments of a common plan. When we are concerned with the links between the main segment and its sub-segments there are two other kinds of data-flow links: Input-Input in which an input of the main segment is passed directly to a sub-segment as one of its inputs, and Output-Output in which the output of a sub-segment is passed to the main segment as one of its outputs.

Control-flow links are included for two purposes. A simple control-flow link states that one segment must finish its execution before the other segment may begin to execute. This is included so that segments with side-effects can be properly ordered to avoid destructive interference. Data-flow links imply an ordering relationship as well, since a segment may not execute until all its inputs have arrived. Other than these constraints there is no ordering imposed; plan diagrams are always interpreted in a (pseudo) parallel manner, even though I use them to analyze the behavior of sequential processes.

The more complicated use of control-flow links is to specify where control will go from a segment whose specs split into cases. Conditional-control-flow links connect a particular case of a segment to its succeeding segment. Thus, if a segment has two cases (a typical test) there will be one conditional-control-flow link for each case leading to that segment which should next be evaluated if that particular case is applicable. A segment which terminates a conditional-control-flow link cannot execute unless the initiating case of the link is applicable.



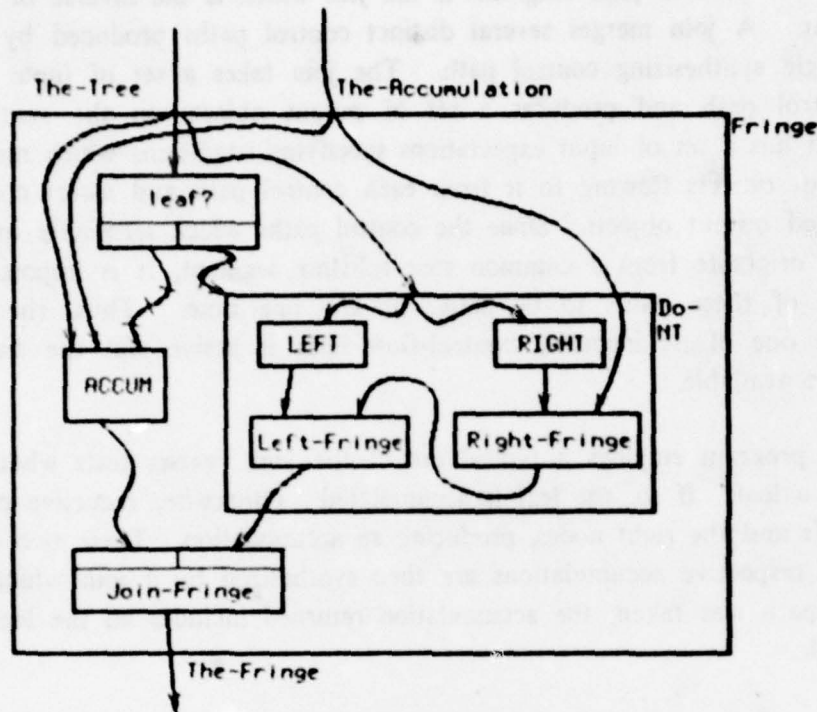
Conditional-Control-flow Links

It should be noted that control-flow links are not connected to a particular port of a segment in the way that a data-flow is. The symbolic interpreter which I will describe in the next chapter interprets the control-flows, since they are extrinsic to the segment. The presence of a control-flow is in no way related to any intrinsic specification of the segment.

One final tool used in plan diagrams is the *join* which is the inverse of a case-splitting segment. A join merges several distinct control paths produced by a case split into a single synthesizing control path. The join takes a set of input objects from each control path and produces a set of output objects on the synthesizing control path. It has a set of input expectations specifying conditions which must hold true of the input objects flowing to it from each control path and assert conditions about the merged output objects. Since the control paths which terminate at a join are required to originate from a common case splitting segment, it is impossible for more than one of these paths to be active at any one time. Thus, the *join* is applicable when one of its incoming control-flow links is active and the associated input objects are available.

The *FRINGE* program employs a typical use of the join. *FRINGE* tests whether the current node is a leaf. If so, the leaf is accumulated. Otherwise, recursive calls are made on the left and the right nodes, producing an accumulation. These two controls paths and their respective accumulations are then synthesized by a join which states that whichever path was taken, the accumulation returned includes all the leaf nodes of the input node.

We may use these notions to represent the fringe program diagrammatically as follows:



Schematic Plan Diagram For Fringe

Notice that in the above diagram, the "railroad track" lines represent control-flow, while the solid lines represent data-flow.

This diagrammatic representation of the program has a direct translation into specifications, data- and control-flow assertions as shown:

```

(dataflow (input fringe the-tree)
  (input leaf? the-node))

(dataflow (input fringe the-tree)
  (input process-non-leaf the-tree))
(dataflow (input fringe the-accumulation)
  (input process-non-leaf the-accumulation))

(dataflow (input fringe the-accumulation)
  (input accumulate-fringe the-accumulation))
(dataflow (input fringe the-tree)
  (input accumulate-fringe the-new-element))

(dataflow (input process-non-leaf the-tree)
  (input left the-tree ))
(dataflow (input process-non-leaf the-tree)
  (input right the-tree ))

(dataflow (output right the-node)
  (input right-fringe the-tree))
(dataflow (input process-non-leaf the-accumulation)
  (input right-fringe the-accumulation))

(dataflow (output left the-node)
  (input left-fringe the-tree))
(dataflow (output right-fringe accumulation)
  (input left-fringe the-accumulation))

(dataflow (output left-fringe accumulation)
  (output process-non-leaf accumulation))

(dataflow (output process-non-leaf accumulation)
  (input (join-fringe case-2 the-accumulation)))
(dataflow (output accumulate-fringe the-accumulation)
  (input (join-fringe case-1 the-accumulation)))

(dataflow (output join-fringe the-accumulation)
  (output fringe the-accumulation))

(conditional-control-flow
  ((test-leaf case-2) process-non-leaf))
(conditional-control-flow
  ((test-leaf case-1) accumulate-fringe))

(control-flow accumulate-fringe (join-fringe case-1))
(control-flow process-non-leaf (join-fringe case-2))

```



```

(spec-type test-leaf leaf?)
(spec-type left-fringe fringe)
(spec-type right-fringe fringe)
(spec-type accumulate-fringe cons)

```

If two segments are internally identical, i.e. if they have identical internal structure then we say that they have the same *plan-type*. It follows that if *SEGMENT-1* and *SEGMENT-2* are of the same *plan-type* then they are also of the same *spec-type*.

A plan type is defined using six clauses

- (i) A list of sub-segment names (we will often refer to these names as roles of the plan).
- (ii) A set of type constraints on the sub-segments, i.e. a list of what *plan-type* or *spec-type* constraints they satisfy
- (iii) A set of data-flow links.
- (iv) A set of control-flow (and conditional-control-flow) links.
- (v) A set of input names.
- (vi) A set of output names.

This may be specified as follows:

```

(defplan fringe
  (sub-segments: test-leaf left-fringe right-fringe
                 left right accumulate-fringe)
  (constraints: (spec-type test-leaf leaf?)
                 (plan-type left-fringe fringe)
                 (plan-type right-fringe fringe)
                 (spec-type accumulate-fringe cons)
                 (spec-type left left)
                 (spec-type right right))
  (flow-diagram: (dataflow ... )
                 ... ))

```

If two segments have the same *plan-type*, then their internal structure is identical to the degree of detail specified in the plan diagram. Thus, their internal temporal behavior is identical. Notice that the plan diagram for the *plan-type* *FRINGE* includes two sub-segments whose *plan-type* is also *FRINGE*. It follows that if we wish to prove some temporal property of the *FRINGE* *plan-type* we may do so by an inductive argument, assuming that this property holds of the two sub-segments of *plan-type* *FRINGE* and deriving the desired property of the main segment. It is important to remember the distinction between *plan-types* and *spec-types*. Knowing a segment's

spec-type will not help in inductive proofs of its internal temporal properties; spec-type, in contrast to plan-type, is strictly concerned with I/O behavior.

Chapter 6: A Symbolic Interpreter for Plan Diagrams

In this chapter I will describe how REASON proves properties of a plan diagram through a process called symbolic interpretation [Rich & Shrobe, 1976], [King, 1976], [Smith & Hewitt, 1975], [Yonezawa, 1977]. This process is extremely thorough, recording all dependencies between the various statements in the plan diagram and in the sub-segments' specs. The mechanisms explained here lay the groundwork for being able to describe the internal temporal behavior of a segment; this will be used in the next chapter where I will develop a more powerful set of descriptive tools used in the process of *plan recognition*.

Recall from Chapter 4 that a *situation* is defined to be a point of time during a computation, and that, in general, facts are true in a particular situation.

(*<Assertion> <Situation>*)
 e.g. ((First list-1 object-1) Situation-1)

An *application* represents the result of applying a segment to a set of input objects which satisfy the expectations of the segment, yielding a set of output objects which satisfy the assert clauses of the segment. An application consists of (i) a segment, (ii) a set of input objects and a mapping of these objects to the input names of the segment, (iii) a set of output objects and a mapping of these objects to the output names of the segment, (iv) an input situation, and (v) an output situation. This is represented as follows:

(i) (Segment-part *<application>* *<segment-name>*)
 (ii) (Input *<application>* *<segment-input-object-name>* *<object>*)
 (iii) (Output *<application>* *<segment-output-object-name>* *<object>*)
 (iv) (Input-situation *<application>* *<situation>*)
 (v) (Output-situation *<application>* *<situation>*)

Each of these relations is a function (i.e. uniquely determines its last argument); we may, therefore, use such descriptions to refer to an object unambiguously, using the bracket notation defined in Chapter 4:

(comes-before [input-situation application-1] situation-5)

which says that the input situation of APPLICATION-1 precedes SITUATION-5.

A plan diagram for a segment determines those applications which will take place during the segment's execution. It also determines the set of input and output situations of these applications. Finally, the plan diagram determines a partial ordering on these situations which represents the minimal ordering constraint on segment execution consistent with correct execution. This information can be made explicit by a symbolic interpretation of the plan diagram.

Given a set of input objects for a main segment we may interpret the plan diagram in the much the same way as a LISP interpreter interprets its code. However, since our concern is with the general behavior of the plan diagram, the input objects will be symbolic values representing typical inputs. Thus, any behavior which can be shown to result from applying the segment to these symbolic objects must necessarily also be true when applied to any actual input.

I have so far shown specs and plan diagrams as "packages", i.e. as a single large set of statements. However, in order to build dependencies correctly, the various sub-parts of these packages must be accessible as individual statements. REASON, therefore, expands plan diagrams and specs into an internal format in which each separate idea is represented as an individual fact. We will see how these are used as I explain the symbolic evaluation process.

The interpretation begins by creating an anonymous object to stand for the current application of the main segment. For simplicity this name is always chosen as the plan-type name of the diagram. REASON then proceeds, assigning the input objects to the appropriate input ports of the main segment. As each object is assigned to an input port, the symbolic interpreter adds an assertion to the data-base stating the assignment. For example, if LIST-1 is the input object matched to the input port THE-CURRENT-LIST of application A-1 then REASON would assert

```
(Input A-1 The-Current-List List-1)
```

The expect clauses are then substituted into, replacing each of the segment's input port names by the actual input object which is assigned to that port. A situation is created to serve as the input situation of the main segment and the substituted expect clauses of the main segment are assumed to hold in this input situation.

For Complex Program Understanding

Each input port of the main segment is connected via at least one data-flow link to an input port of some sub-segment. Intuitively, the data-flow link transports the object from the specified port of the main segment to that of the sub-segment. Whenever the symbolic evaluator sees that an object is bound to a port which is connected to the initiating side of a data flow link, it simulates the data-flow by assigning the same object to the port which terminates the data-flow. The justification for the assertion stating this assignment points to the data-flow link and the assertion stating the assignment of the object to the initiating port. For example, if we had the following:

```
F-1      (Input A-1 The-Current-List List-1)
F-2      (Segment-Part A-2 Sub-Seg-2)
F-3      (Data-flow (Input A-1 The-Current-List)
              (Input Sub-Seg-2 The-New-List))
```

then REASON would assert:

```
F-4      (Input A-2 The-New-List List-1) (dflow F-1 F-2 F-3)
```

When all of a sub-segment's input ports have been assigned input objects the sub-segment is ready for application, an application name is created and asserted to be the application name of the current invocation of the sub-segment.

If the segment's spec-type is provided in the plan diagram then application proceeds as follows: First, a situation is created to serve as the input situation of the sub-segment, for example:

```
F-5      (Input-Situation A-2 Situation-2-In)
```

Next, since the expect clauses of the segment's specs are required to be true in this input situation, REASON creates a goal to show that each expect clause holds. These goals are the expect clauses with the input objects substituted for their corresponding input names. Each such goal assertion has a dependency pointing to (i) all of the INPUT assertions relevant to that clause, (ii) the spec clause from which the goal is built, and (iii) the assertion stating the input situation of the application. If all of the goals are satisfied the segment is applicable, otherwise the plan has an error. The assertion that the segment is applicable is justified by a dependency which points to the satisfied assertions for each expect clause goal. Thus, if sub-seg-2 has expects $\epsilon-1$ and $\epsilon-2$ and if sub-seg-2 has spec-type spec-3 then the following assertions are created:

```

F-6   (Spec-type Sub-Seg-2 Spec3)
F-7   (Spec-Clause Spec3 Expect Case-0 Clause-1
      (E-1 The-New-List))
F-8   (Spec-Clause Spec3 Expect Case-0 Clause-2
      (E-2 The-New-List))

F-9   (Goal ((E-1 List-1) Situation-2-In)      (ExpClause F-7 F-6 F-5 F-4 F-2)
      for (Expect-clause-of A-2) in ())

F-10  (Goal ((E-2 List-1) Situation-2-In)      (ExpClause F-8 F-6 F-5 F-4 F-2)
      for (Expect-clause-of A-2) in ())

```

notice the use of the "for" part of the goal assertion to indicate that the higher level task from which the goal arose is the symbolic interpreter's expect checking routine.

When these goals are satisfied we obtain the following:

```

F-20  (satisfied (goal ((E-1 List-1) Situation-2-In)
                    for (Expect-clause-of A-2) in ()))

F-21  (satisfied (goal ((E-2 List-1) Situation-2-In)
                    for (Expect-clause-of A-2) in ()))

F-22  (Applicable A-2) (expects-satisfied F-21 F-20)

```

When the segment is shown to be applicable a new situation is created to serve as the segment's output situation. If the segment's specs specify that any of the outputs are new objects (i.e. created within this segment's execution), then the interpreter creates object names for these outputs and assigns them to the appropriate output ports. This is done using an OUTPUT assertion which is similar to that INPUT assertion shown above. The assert clauses of the specs are instantiated, replacing each (input or output) port name by the name of the object assigned to that port. The appropriate situational tags are also defaulted into these assertions. These instantiated assertions are then asserted with justifications which point to the statement that the segment is applicable, to the spec-type assertion for this segment, to the assertion representing the actual spec clause from which this assertion is built, and to the relevant INPUT and OUTPUT assertions. Suppose that A-2 is determined to be applicable and that it produces an output named OUT-LIST. Suppose, further, that the assert clauses of this segment specify that its output is to be sorted. Then, REASON would create the new object OUT-LIST-1 to stand for this output, asserting:

For Complex Program Understanding

110 A Symbolic Interpreter for Plan Diagrams

F-40 (Output-Situation A-2 Situation-2-Out)

F-41 (Output A-2 Out-List Out-List-1)

F-42 (Spec-clause Spec3 Assert Case-0 Clause-1 (Sorted Out-List))

F-43 ((Sorted Out-List-1) Situation-2-Out) (Output-Assert F-22 F-42 F-41 F-40 F-6 F-2)

The output ports of the sub-segment just interpreted are linked to other segments via data-flow links. These may terminate at either input ports of other sub-segments or at output ports of the main segment. If the data-flow link terminates at another sub-segment's input port, the object assigned to the output port of the first segment is then assigned to the input port of the second. This process produces assertions like those created by the data-flows from the main segment's to sub-segment's input ports. For example, if the output of A-2 flows to SUB-SEG-3's SORTED-LIST input we would get the following assertions:

F-100 (Dataflow (Output Sub-Seg-2 Out-List)
(Input Sub-Seg-3 Sorted-List))

F-101 (Segment-Part A-3 Sub-Seg-3)

F-102 (Input-Situation A-3 Situation-3-In)

F-103 (Input A-3 Sorted-List Out-List-1) (dflow F-41 F-2 F-100 F-101)

If the terminating sub-segment now has all of its input objects bound, it is ready for application and we proceed as above.

If a data-flow link leads from a sub-segment output port to an output port of the main segment, then the object assigned to the sub-segment's port is transported to the output port of the main segment. Assertions and justification like those above are created. If object are assigned to all the output ports of the main segment, then interpretation of the plan diagram is complete and an output situation for the main segment is created. If the plan diagram correctly implemented the specs of the main segment, then the assert clause of the main segment should be provable in its output situation. Goal statements like those created for expect clauses of a sub-segment are created for the assert clauses of the main segment; these goals are justified in a manner similar to that above.

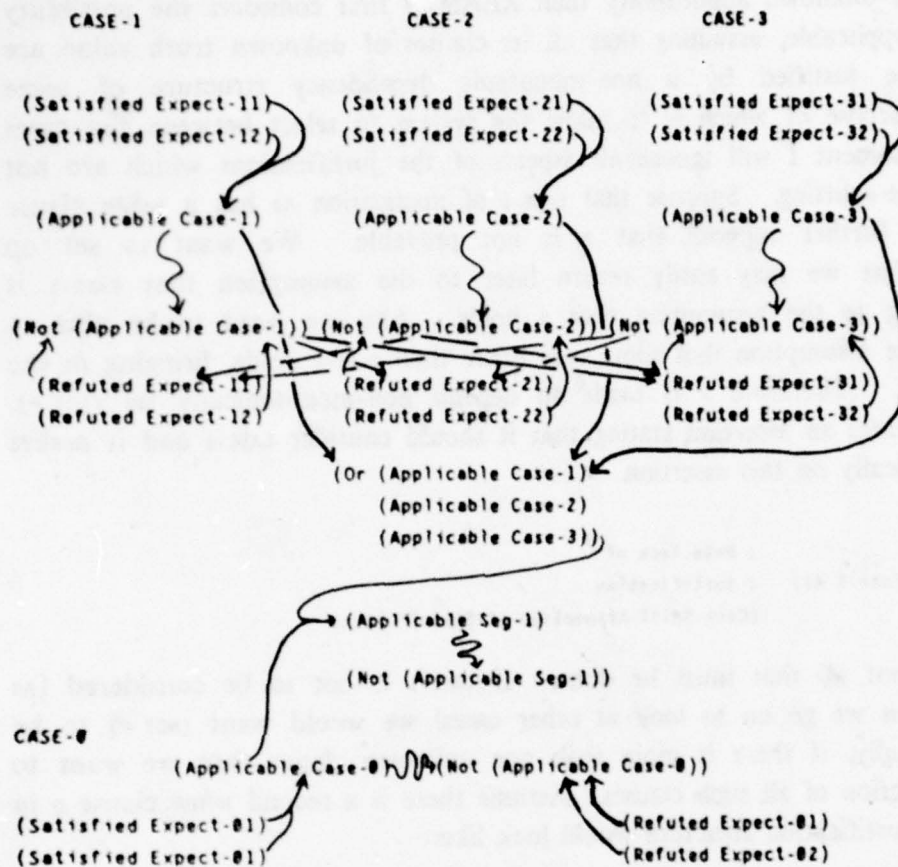
So far I have assumed that the spec type of each sub-segment is known. However, if the segment's spec-type is not known, but rather an internal plan diagram is provided (i.e. we know its plan but not its specs), then the sub-segment is interpreted recursively; an output situation and a set of output objects will result. The interpretation then continues as above. If both the specs and the plan diagram for a sub-segment are known, REASON first uses the specs in interpreting the outer diagram and then returns to the inner diagram, symbolically interpreting it and showing that its specs follow from its plan diagram. As we will see later, this allows us to break the task up into smaller pieces; in the case of recursions and loops it provides a means for stating a "subgoal invariant" [Morris & Wegbreit, 1977].

A subsegment which has cases presents additional complexity. Like other segments the segment with case-splits may have expect clauses which must be true for the segment to function and asserts which are true no matter which case is taken. These are called the *case-0* clauses; if the case-0 expects cannot be proven in the segment's input situation an error has been detected. If the case-0 expects are shown to be satisfied, the case-0 asserts may be asserted in the output situation. However, once the case-0 clauses have been proven it is still necessary to show that at least one of the other cases is valid. This is done by iterating over the cases attempting to prove the *when* clauses of each. As each case is attempted, REASON sets the goal of showing that each *when* clause of the case is provable. If these goals are satisfied, the case is *applicable*. Before starting however, it assumes that the case is *inapplicable* so that unless a proof of applicability is found, REASON prudently assumes that there are no applicable cases.

Each attempted proof of a *when* clause can lead to one of three results: a proof of the clause could be found, a refutation of the clause could be found, or neither of the above. In many practical cases, the system will be able to know when it has reached a case of unprovability; typically the inputs to the main segment of a plan diagram are not highly constrained by the expect clauses; for example, a segment might expect a list as input and then test internally for emptiness, taking different branches for the two possibilities. In such cases the system can determine that it cannot know whether the input list is null or not; instead REASON will immediately engage in a case-split analysis. This avoids the wasted effort of attempting to make impossible proofs.

If proofs can be found for all the clauses of a case, then the case is *applicable*. REASON asserts the case to be applicable with a justification pointing to those facts which satisfied the *when* goals. The assert clauses of the case are added to the output situation and justified as above. If a *when* clause of a case is refuted, the case is declared to be *inapplicable* with a justification pointing to the refuting fact. The next case is then tried.

The specs used in REASON for case-splitting segments assumes that the cases are ordered sequentially, i.e. CASE-2 can only be considered if CASE-1 is inapplicable, and similarly for the remaining cases. Thus, the goal of showing the applicability of CASE-2 includes a subgoal that CASE-1 is not applicable. CASE-3 includes the two goals that CASE-1 and CASE-2 are not applicable. However, REASON will not attempt a case unless it already knows that the prior cases are not applicable (or unless they are of *unknown applicability*, as I will discuss next). Therefore, REASON already knows the results for the previous cases and includes these in the dependencies supporting the assertion which invoked the conjunctive goal mechanism. This builds up a justification structure which guarantees that no more than one case can have its applicable assertion *in* at the same time. Thus, a segment with three cases would have justifications like the following (Note: wavy lines indicate non-monotonic dependency).



Support Structure For Case-Splitting Segment

Notice that this is an "and-or" graph. Individual justifications include dependence on several facts at once (conjunction is indicated by lines joining together at an arrow) while several justifications may independently support the same fact (disjunction is indicated by separate arrows pointing at the same fact). Also notice that the dependencies guarantee that at most one case will be considered applicable at a time.

However, the normal circumstance is that each case (except the last) will have some *when* clause which can be neither proved nor refuted; we then say that the clause is of *unknown truth value*. A case which contains no refuted clauses and at least one clause of unknown truth value is said to be of *unknown applicability*. These cases reflect the possibility that a test might sometimes succeed and sometimes fail, depending on the input data. Since we are interpreting the plan on symbolic (typical) inputs, most segments with case structure will have cases of unknown applicability.

For Complex Program Understanding

114 A Symbolic Interpreter for Plan Diagrams

If a case has unknown applicability then REASON first considers the possibility that the case is applicable, assuming that all its clauses of unknown truth value are true. These are justified by a non-monotonic dependency structure of some complexity, the purpose of which is to allow the system to select between the cases later. For the moment I will ignore all aspects of the justifications which are not related to the case-splitting. Suppose that CASE-1 of application A3 has a *when* clause requiring P, and further suppose that P is not provable. We want to set up justifications so that we may easily return later to the assumption that CASE-1 is applicable, bringing *in* the assumption that P holds. Also we want to be able to switch easily to the assumption that some case other than CASE-1 holds, bringing *in* the assumption (NOT P). Therefore P is made to depend non-monotonically on (NOT P). Also REASON creates an assertion stating that it should consider CASE-1 and it makes P depend monotonically on this assertion.

```
F-110      (not P)           ; Note lack of
F-111      (Select Case-1 A3) ; justification
F-112      P                (Case-Split-Assumption (F-111) (F-110))
```

However, this is not all that must be done. If CASE-1 is not to be considered (as would happen when we go on to look at other cases) we would want (NOT P) to be brought *in*. Actually, if there is more than one unknown clause then we want to bring in the disjunction of all such clauses. Assume there is a second *when* clause Q in CASE-1. Then the justification structure would look like:

```
F-110      (not P)           ; no justification
F-111      (Select Case-1 A3) ; no justification
F-112      P                (Case-split-Assumption (F-111) (F-110))
F-113      (not Q)          ; no justification
F-114      Q                (Case-split-Assumption (F-111) (F-113))
F-115      (Or (not P)(not Q)) (Case-Split-Assumption (F-116) (F-111))
F-116      (Select Case-2 A3) ; no justification
```

If there is a third case, then F-115 should also be brought *in* whenever this case is being considered. REASON adds another justification to F-115 as follows:

```
F-117      (Select Case-3 A3) ; no justification
F-115      (Or (not P)(not Q)) (Case-Split-Assumption (F-117) (F-111))
```

REASON may now consider any case simply by giving the appropriate (SELECT CASE-1 A3) assertion a justification. When it is through considering the case, it must remove the justification for the *select* assertion by retracting the justification supporting the *select* assertion. Notice that if no *select* assertion is justified, then none of the assumptions are *in*, representing the most general case where we have no idea which case holds.

Once REASON has assumed all clauses of *unknown truth value* for a particular case, it will have satisfied all the case's *when* clauses. The case will then be declared *applicable* and a justification created pointing to all assertions satisfying any of the *when* clauses, including the assumptions justified by the *select* assertion. The output clauses of the case are then asserted in the output situation, each being justified by the assertion declaring the case applicable. Thus, the logical relationship between the assumptions and the output assertions is represented explicitly in the data-base. Finally, if there are conditional-control-flow links coming from the current case, these are declared active with a justification pointing to the assertion which declares the case applicable. REASON now continues evaluating any segments which terminate the conditional-control-flow links leaving the segment until it reaches either a JOIN or the output side of the main segment of the plan diagram.

As REASON goes along the paths started by the conditional-control-flow links, it records which cases have yet to be evaluated. Thus, when a terminal segment is reached it returns to evaluate the remaining cases.

It begins by *outing* the *select* assertion for the last case evaluated, removing the assertion's justification. It then justifies the *select* assertion for the next case. This has the effect of assuming the falseness of at least one of those clauses from the previous case which had unknown truth value. The inapplicability of the previous case follows from this assumption directly (using proof by cases if there is more than one clause of unknown truth value); REASON constructs this proof recording the appropriate justifications. Thus, selecting CASE-2 will *out* the *applicable* assertion of CASE-1.

The next case is then evaluated with its *select* assertion *in*, i.e. REASON investigates whether the next case's applicability follows from the inapplicability of the previous case. If the new case has clauses of unknown truth value, then REASON proceeds as above, creating non-monotonic justifications for the current case's applicability. If all the clauses of the new case can be proven, then this case is

For Complex Program Understanding

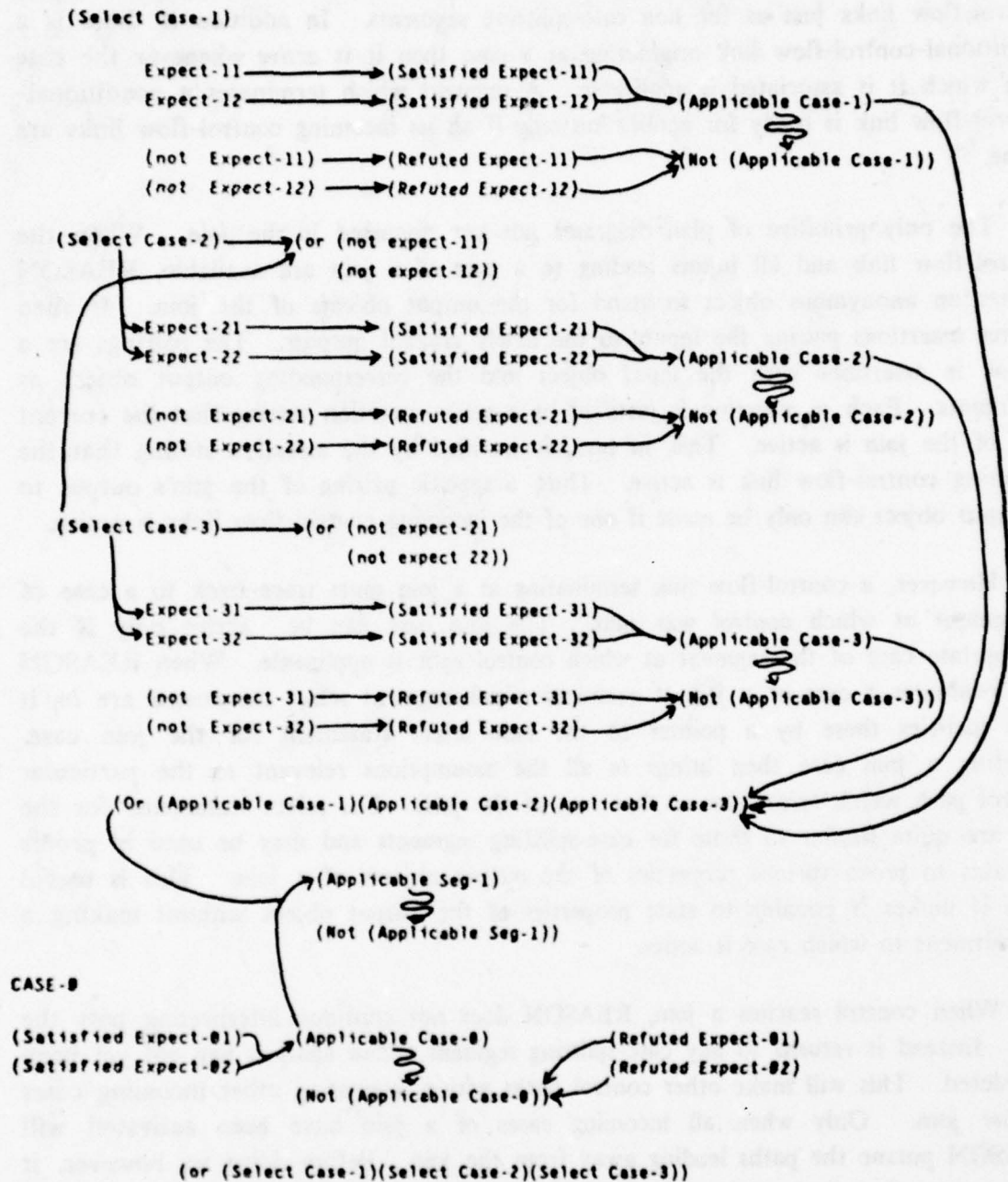
declared applicable. If the case has a clause which is definitely false, the case is declared inapplicable; REASON then moves on to the next case.

If the last case has clauses of unknown truth value, then the segment is declared inapplicable with a justification pointing at a statement expressing the possibility that the unproven clauses might be false. This statement is justified so that it depends on the *outness* of the clauses of unknown applicability; if something is changed to make these clauses definitely true, the inapplicable assertion will go *out*. If the last case has clauses of unknown truth value, then there is an error which manifests itself as an intermittent program bug; if the input data happened to be such that an earlier case would succeed then the program would work, otherwise it would fail. Such intermittent bugs are among the most distressing problems of programming and it is desirable to be able to spot them through the process of symbolic evaluation.

Since REASON requires both that at least one case be applicable and that *case-0* be applicable in all circumstances, the final action taken in evaluating a case-splitting segment is to assert the disjunction of all the *select* assertions and to attempt to prove the *applicable* assertion for the segment. This is always done by case-splitting the disjunction of the select assertions. Thus if there were three cases, we would have:

```
F-500 (Or (Select Case-1 A3)(Select Case-2 A3)(Select Case-3 A3))
F-501 (Subgoal (And (Or (Applicable Case-1 A3) (Applicable Case-2 A3) (Applicable Case-3 A3))
                  (Applicable Case-0 A3))
        for ((Applicable A3) ...)
        in (...))
F-502 (Show (goal (Or (applicable ...)))
        by (splitting (OR (select ...)))
        for ((Applicable A3) ...) in (...))
```

This proof proceeds trivially; all the justifications have already been built up. As each select statement is assumed by the case-splitting mechanism, the corresponding case becomes applicable and the disjunction in *r-501* is deduced by disjunction introduction. If the last case had not been found applicable, however, then the appropriate clause will not come *in* and the goal will not be deducible. REASON complains that it has found a bug. In any event the final justification structure built up in this process looks as follows:



Support Structure For Case-splitting With Clauses of Unknown Truth Value

For Complex Program Understanding

When a case is declared applicable its output objects are propagated along control-flow links just as for non case-splitting segments. In addition if there is a conditional-control-flow link originating at a case then it is *active* whenever the case with which it is associated is applicable. A segment which terminates a conditional-control-flow link is ready for application only if all its incoming control-flow links are active.

The only primitive of plan diagrams not yet discussed is the *join*. When the control-flow link and all inputs leading to a case of a join are available, REASON creates an anonymous object to stand for the output objects of the join. It then creates assertions pairing the inputs to the newly created outputs. The pairings are a set of *n* assertions with the input object and the corresponding output object as arguments. Each *n* assertion is justified by a *SELECT* assertion stating that the current case of the join is active. This, in turn, is justified by the assertion stating that the incoming control-flow link is active. Thus, a specific pairing of the join's output to an input object can only be made if one of the incoming control-flow links is active.

However, a control-flow link terminating at a join must trace back to a case of a segment at which control was split. The join case can be active only if the appropriate case of the segment at which control split is applicable. When REASON first evaluates a case of a join it examines which segment select statements are *in*; it then justifies these by a pointer to the case select statement for the join case. Selecting a join case then brings *in* all the assumptions relevant to the particular control path which terminates at that case of the join. The select statements for the join are quite similar to those for case-splitting segments and may be used in proofs by cases to prove various properties of the output objects of a join. This is useful since it makes it possible to state properties of the output object without making a commitment to which case is active.

When control reaches a join, REASON does not continue interpreting past the join. Instead it returns to any case-splitting segment whose analysis has not yet been completed. This will make other control paths active, activating other incoming cases of the join. Only when all incoming cases of a join have been activated will REASON pursue the paths leading away from the join. Before doing so, however, it makes sure that it has cleaned up the evaluation of all prior case-splits so that evaluation of the paths leading out from the join does not inadvertently proceed under the assumption that only a single case of the case-splitting segment need be considered.

When all interpretation is completed REASON attempts to prove that the assert clauses of the main segment are satisfied. Again each clause is translated into a goal and the reasoning mechanisms of the previous chapter are invoked. If the proofs succeed, justifications are built as before. Thus, when all the goals are proved a complete dependency network is built, linking every satisfied goal back to the primitives of the plan diagram upon which the goal depends. These dependencies point to data-flow and control-flow links, to input assertions of the main segment and to output assertions of the sub-segments.

Each such proof of a goal can be categorized as either a pre-requisite proof or an achieve proof. Pre-requisite proofs are those which establish that a sub-segment's *expect* and *when* clauses are satisfied. Achieve proofs are those proving the assert clauses of the main segment. If these are summarized to remove the detail, leaving only the connections between specs clauses and flow links then we have what we have called *purpose links*.

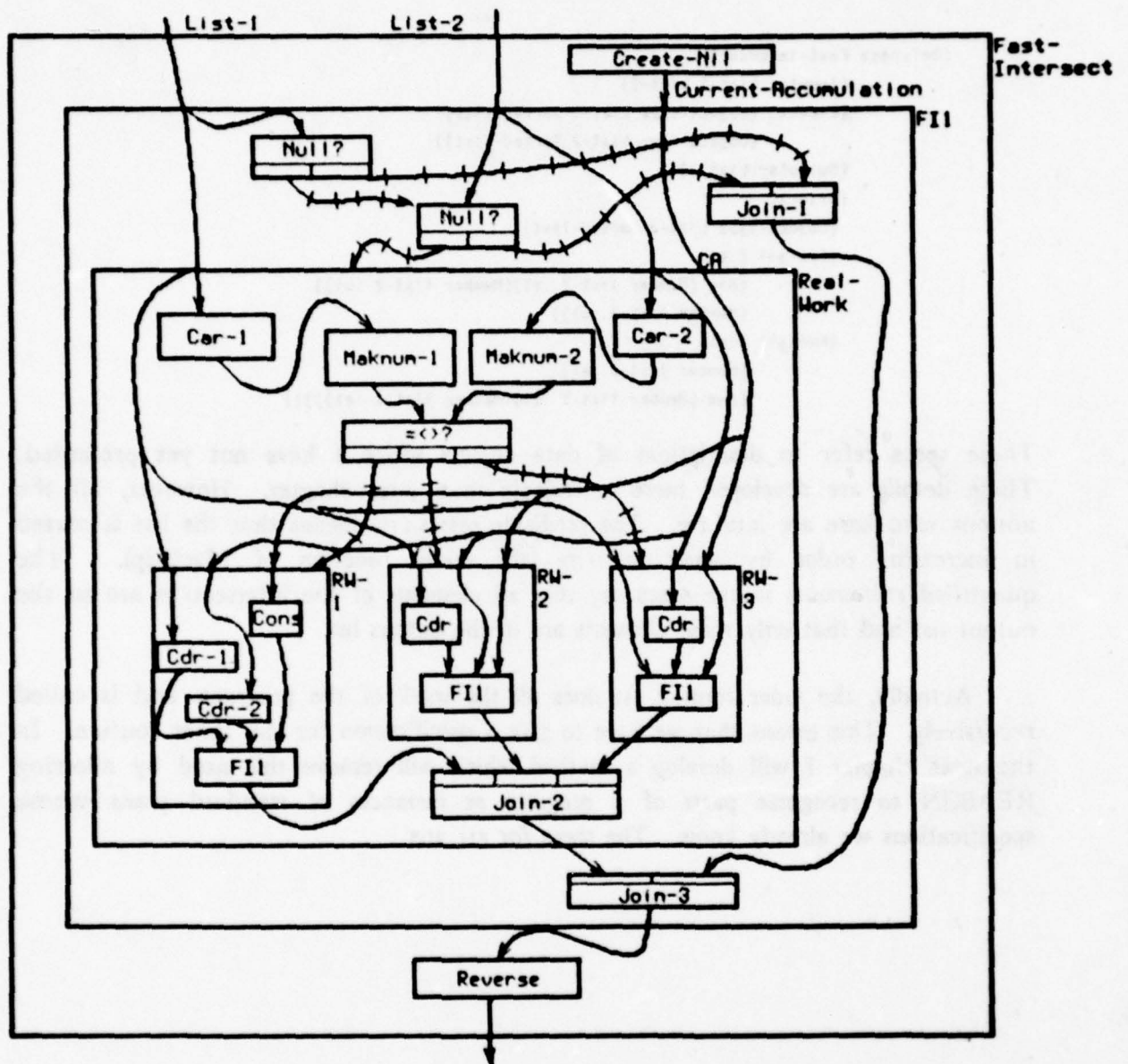
We see, therefore, that a symbolic interpretation in REASON leads to considerably more information than just the statement that the program does what is intended. In addition to this data, REASON produces a complete proof and a summary of this proof into purpose links which quickly indicate the intermodule dependencies in the program. Furthermore, this data is so organized that if a crucial spec clause is changed then all other sub-segments which depended on this clause will be declared inapplicable. This change of spec status will be signalled by the Truth Maintenance System as part of its normal *iming* and *ouing* of facts. REASON responds to these notices and informs the user of the exact nature of the problem caused by the change. Purpose links provide a rapid mechanism whereby REASON can tell without a deeper analysis that a proposed change is not safe. Since the purpose links tell whether a spec clause is used in any proof, all REASON must do is to see if the clause is involved in any purpose links. If so, the link tells which segments are affected by the change.

Chapter 7: An Example of Symbolic Interpretation

To show how the features developed so far interact in the analysis of a moderately complicated algorithm, let us consider how REASON interprets a routine for computing the intersection of two sets represented as ordered lists. This algorithm runs in linear time by only considering the heads of both lists. If the two heads are identical, then that element should be added to the accumulation. If they are not identical, then the smaller element cannot also be a member of the other list. Thus, it can be thrown away and the iteration continued. One possible coding of this routine is as follows:

```
(Defun fast-intersect (list-1 list-2)
  (do ((Acc nil)
      (Car-11 Nil)
      (Car-12 Nil)
      (Uid-1 Nil)
      (Uid-2 Nil))
    ((Or (Null list-1) (Null list-2))
     (Reverse Acc))
    (Setq Car-11 (car list-1) Car-12 (car list-2))
    (Setq Uid-1 (maknum car-11) Uid-2 (maknum car-12))
    (Cond
      ((Eq Uid-1 Uid-2) (Setq Acc (cons Car-11 Acc))
                        (Setq list-1 (cdr list-1)
                             list-2 (cdr list-2)))
      ((< Uid-1 Uid-2) (Setq list-1 (cdr list-1)))
      (t (Setq list-2 (cdr list-2)))))
```

As we mentioned before, the LISP code is first analyzed by a surface flow analyzer which abstracts out many of the details of surface data and control-flow. In particular, SETQ's used to achieve data-flow are translated into data-flow links and COND's and other control primitives are translated into case structured segments with conditional-control-flow links. The plan diagram given to REASON for analysis is the following:



For Complex Program Understanding

122 An Example of Symbolic Interpretation

The specs for this routine are as follows:

```
(Defspecs Fast-intersect
  (Inputs: List-1 List-2)
  (Expect: (Object-type List-1 Sorted-list)
            (Object-type List-2 Sorted-list))
  (Outputs: List-3)
  (Assert:
    (Object-type List-3 Sorted-list)
    (For-all (:el)
      (And (Member list-1 :el)(Member list-2 :el))
      (Member list-3 :el)))
  (For-all (:el)
    (Member list-3 :el)
    (And (Member list-1 :el)(Member list-2 :el))))
```

These specs refer to descriptions of data objects which I have not yet presented. These details are developed more extensively in a later chapter. However, all the notions used here are intuitive. The predicate `SORTED-LIST` means that the list is sorted in increasing order by `UNIQUE-IDENTIFIER` (the `MAXNUM` function of MacLisp). The quantified statements in the specs say that all elements of the intersection are in the output list and that only these elements are in the output list.

Actually, the inner routine `F11` does all the work of the program, and is called recursively. This means that we have to give a specification for this inner routine. In the next chapter I will develop a method which will remove this need by allowing `REASON` to recognize parts of a program as instances of standard plans whose specifications we already know. The specs for `F11` are:

```

(defspecs F11
  (Inputs: List-1 List-2 CA)
  (Expect: (Object-type List-1 Sorted-List)
            (Object-type List-2 Sorted-List)
            (Object-type CA Reverse-Sorted-List)
            (For-all (:x)
                      (Or (Member List-1 :x)(Member List-2 :x))
                      (For-all (:y)
                                (Member CA :y)
                                (< [Unique-Id :y] [Unique-Id :x])))))
  (Outputs: Final-Accum)
  (Assert:
    (Object-type Final-Accum Reverse-Sorted-List)
    (For-all (:x)
              (And (Member List-1 :x)(Member List-2 :x))
              (Member Final-Accum :x))
    (For-all (:x) (Member CA :x)
              (Member Final-Accum :x))
    (For-all (:x) (Member Final-Accum :x)
              (Or (and (Member List-1 :x)
                      (Member List-2 :x))
                  (Member CA :x)))))

```

Given these specs for F11, it is an immediate consequence that FAST-INTERSECT satisfies it specs. FAST-INTERSECT calls F11 with its two input lists as the two lists, and with NIL as the CA input. Since nothing is a member of NIL, the second quantified statement in the asserts of F11 is vacuous; similarly the third quantified statement contains a disjunction whose second disjunct is vacuous if the CA input is NIL, the other disjunct is exactly that required by FAST-INTERSECT. Similarly, the expect clauses are met simply; since the input CA is NIL, it is a vacuous condition that all elements of LIST-1 and LIST-2 have larger uid's than the elements of CA. Finally, F11 produces a list in reverse sorted order which is then reversed by FAST-INTERSECT, producing the required sorted list as output.

I will now describe the actions which the symbolic interpreter takes in evaluating the above plan diagram. However, going through all of the details would be an overly cumbersome exercise, so I will try to present this without too much tedium and repetition. The system begins by creating an input situation and anonymous objects to stand for the inputs to the program. We will call these S-IN, LIST-1 and LIST-2 respectively. The system then asserts the expect clauses of FAST-INTERSECT's specs. This gives us:

For Complex Program Understanding

124 An Example of Symbolic Interpretation

F-1 ((Object-type List-1 Sorted-List) S-In)
 F-2 ((Object-type List-2 Sorted-List) S-In)

Next the system evaluates the segment CREATE-NIL which simply asserts that its output is NIL: REASON names the output situation of CREATE-NIL CREATE-NIL-OUT, thus we have:

F-4 (Output Create-nil The-Null-Object Nil)
 F-5 ((Object-type Nil Empty-List) Create-Nil-Out)

The data objects are next moved along the data-flow links to the input ports of F11, the routine which actually does the work. Since there are no side-effects in this program, all assertions which are true in one situation will be true in all succeeding situations. (Side-effects change this drastically; I will discuss the problems of side-effects in greater detail later). Therefore the following facts are true in the input situation of F11 which REASON names S-IN-1:

F-6 (Input F11 L1 List-1)
 F-7 (Input F11 L2 List-2)
 F-8 (Input F11 CA Nil)
 F-9 ((Object-type List-1 Sorted-List) F11-In)
 F-10 ((Object-type List-1 Sorted-List) F11-In)
 F-11 ((Object-type Nil Empty-List) F11-In)
 F-12 ((Object-type Nil List) F11-In) (Nil-Is-List F-11)

As mentioned above, REASON has the specs for F11 already. So if it can show that the expects of F11 are satisfied, it can use the asserts directly. These expects are, however, direct conclusions. REASON declares this invocation of F11 applicable, create an output situation F11-OUT, and adds the asserts to this situation, getting:


```

F-15      (Output Fil Final-Accum Final-Accum-0)
F-16      ((Object-type Final-Accum-0 Reverse-Sorted-List) Fil-Out)
F-17      (For-all (:x)
           ((Member NIL :x) Fil-IN)
           ((Member Final-Accum-0 :x) Fil-Out))
F-18      (For-all (:x)
           (And ((Member List-1 :x) Fil-IN)
                ((Member List-2 :x) Fil-IN))
           ((Member Final-Accum-0 :x) Fil-OUT))
F-19      (For-all (:x)
           ((Member Final-Accum-0 :x) Fil-OUT)
           (Or (and ((Member List-1 :x) Fil-IN)
                    ((Member List-2 :x) Fil-IN))
               ((Member NIL :x) Fil-IN)))

```

The outputs now flow to the reverse segment whose only effect is to change the object type statement above, producing a sorted list instead of a reverse sorted list. REASON can then immediately show that the desired results hold in the output situation of FAST-INTERSECT.

However, to use the specs of the internal routine `Fil` with confidence, REASON must demonstrate that its specs follow from its plan diagram; therefore, it creates anonymous inputs for `Fil` and begins to symbolically evaluate the plan diagram for `Fil`. REASON names the two lists input to `Fil` `LIST-1` and `LIST-2` (as above) and the current accumulation `CA`. The expect clauses of `Fil` are asserted in the input situation of the current application of `Fil`:

```

F-20      ((Object-type List-1 Sorted-List) Fil-IN)
F-21      ((Object-type List-2 Sorted-List) Fil-IN)
F-22      ((Object-type CA Reverse-Sorted-List) Fil-IN)
F-23      (For-all (:x)
           (Or ((Member List-1 :x) Fil-IN)
               ((Member List-2 :x) Fil-IN))
           (For-all (:y)
               ((Member CA :y) Fil-IN)
               ((< [Unique-Id :y] [Unique-Id :x]) Fil-IN)))

```

Notice that situation tags have been added to the quantified statements in the spec clauses using the simple defaulting rule that clauses mentioning output objects are assigned to the output situation of the segment. REASON draws a few direct conclusions from the above assertions:

For Complex Program Understanding

126 An Example of Symbolic Interpretation

```

F-25      ((Object-type List-1 List) F11-IN) (type-inherit F-20)
F-26      ((Object-type List-2 List) F11-IN) (type-inherit F-21)
F-27      ((Object-type CA List) F11-IN)      (type-inherit F-22)

```

REASON now begins the symbolic evaluation of F11. The data-flows lead to the two tests which must be evaluated immediately upon entrance to F11. The first of these segments tests whether LIST-1 is null. REASON concludes that there is no relevant information in the situation F11-IN. It creates a case-split, assuming in one case that the list is null and in the other that it is non-null. In the non-null case it must evaluate the second test segment where a similar decision is made. REASON now has three conditional-control-flows waiting for further evaluation. The first of these represents the case where LIST-1 is null. The second represents the case where neither LIST-1 nor LIST-2 is null. The final case is where LIST-2 is null but LIST-1 is not. However, the first and the third cases both lead to JOIN-1. Both cases of JOIN-1 take the same input CURRENT-ACCUMULATION. Since this input is available the join can be evaluated immediately. The only action following from the join JOIN-1 is a second join JOIN-3. This join, however, cannot be evaluated yet since it has another input which is not available. REASON, therefore, returns to the top of the diagram considering the case where both lists are non-null. This case leads to the sub-segment labeled REAL-WORK in the diagram.

REAL-WORK is invoked after both tests have taken the non-null branch. REASON brings in the assumptions of this case, making the following assertions active:

```

F-30      ((Not (Object-type List-1 Empty-List)) REAL-WORK-IN)
F-31      ((Not (Object-type List-2 Empty-List)) REAL-WORK-IN)

```

There are now four segments which may be evaluated immediately: CAR-1, MAKNUM-1, CAR-2, and MAKNUM-2. The two CAR segments create output objects representing the first objects of each list, while the two MAKNUM segments create objects representing the numbers which are the UNIQUE-ID's of the first elements of the two lists. This leads to the following assertions:

```

F-40      ((First List-1 First-1) CAR-1-OUT) ; justifications here indicate
F-41      ((First List-2 First-2) CAR-2-OUT) ; the fact name of the correct
F-42      ((Unique-id First-1 Number-1) MAKNUM-1-OUT) ; spec clauses from
F-43      ((Unique-id First-2 Number-2) MAKNUM-2-OUT) ; each segment's specs.

```

I should explain that my notation in the plan diagram has been somewhat sloppy. CAR-1 and CAR-2 are both segments of spec-type CAR; it is the specs for this spec-type which REASON uses and similarly, for MAKNUM-1 and MAKNUM-2. I should also note at this point that the system makes use of two properties of UNIQUE-ID's which I have not yet stated. First, UNIQUE-ID's are a one-to-one mapping, so that if the UNIQUE-ID of one object is equal to the UNIQUE-ID of a second object, then the two objects are identical. Secondly, since UID's are numbers; any two UID's are either equal, or one of the two is greater than the other.

Following the evaluation of the MAKNUM segments, the only segment ready for evaluation is the test segment labeled *C? which takes as inputs NUMBER-1 and NUMBER-2, anonymous objects representing the UID's of the first elements of the two lists. The test has three cases, corresponding to the possibility that the two numbers are equal, that the first is smaller, or that the second is smaller. REASON decides that there is no evidence available to decide this question and, therefore, creates a case-split. REASON considers the first case first getting the following justification structure:

```
F-98 ((Not (Equal Number-1 Number-2)) *C?-Out) : note no
F-99 (Select Case-1 *C?) : justification
F-100 ((Equal Number-1 Number-2) *C?-Out) (Case-split-assumption (F-99) (F-98))
```

This triggers REASON to conclude that FIRST-1 and FIRST-2 are identical:

```
F-101 (Id First-1 First-2) (One-to-One F-100)
```

The conditional-control-flow link leading from the test *C? leads to the segment labeled RW-1. The data-flows take the object FIRST-1 to the CONS segment as one input; the object CA is the other input. LIST-1 and LIST-2 flow to the two CAR segments. The specs of CONS say that it produces a new CONS-CELL whose left is the object FIRST-1 and whose right is the object bound to CA which is known to be a list. Rules in the system which represent the definition of list membership make several inferences from these two assertions:

128 An Example of Symbolic Interpretation

```

F-199 (Output Cons-1 The-new-cons C-1)      : justifications pointing
F-200 ((Left C-1 First-1) CONS-1-OUT)        : to the spec clauses
F-201 ((Right C-1 Ca) CONS-1-OUT)            : would go here - they are
F-202 ((Object-type C-1 Cons-cell) CONS-1-OUT) : omitted for simplicity

F-203 ((Object-type C-1 List) CONS-1-OUT)    (list-rep f-202 f-201 f-200 f-24)
F-204 ((First C-1 First-1) CONS-1-OUT)      (list-rep f-203 f-200)
F-205 ((Rest C-1 Ca) CONS-1-OUT)            (list-rep f-203 f-201)
F-206 ((Member C-1 First-1) CONS-1-OUT)     (list-mem f-203 f-204)
F-207 (For-all (:x)                        (list-mem f-203 f-205)
      ((Member Ca :x) CONS-1-IN)
      ((Member C-1 :x) CONS-1-OUT))

```

The origin of the rules which make such inferences will be explained in more detail in the chapter on describing data objects.

The two CDR segments produce the obvious output assertions:

```

F-207 ((Rest List-1 Rest-1) CDR-1-OUT)
F-208 ((Rest List-2 Rest-2) CDR-1-OUT)

```

These objects now flow to the recursive call to `F11`. So far I have not mentioned any checking of input expectations since these have all been trivial. `F11`, however, requires that its two input lists be sorted, and that its `ACCUMULATION` input be sorted in reverse order. These first two requirements are met simply; since `REST-1` and `REST-2` are the CDR's of sorted lists, they themselves are sorted. The condition that `C-1` be sorted in reverse order is also met quite simply. Its CDR `CA` is a reverse-sorted list and input expectations stated that `FIRST-1` has a larger `MAXNUM` than any member of `CA`. Thus, `F11` is applicable and its output assertions can be added to the data base. This creates a new output `FINAL-ACCUM-1` and three quantified statements:

```

F-209 ((Object-type Final-Accum-1 Reverse-Sorted-List) Fill-1-Out)
F-210 (For-all (:x)
      ((Member C-1 :x) Fill-1-IN)
      ((Member Final-Accum-1 :x) Fill-1-Out))
F-211 (For-all (:x)
      (And ((Member Rest-1 :x) Fill-1-IN)
            ((Member Rest-2 :x) Fill-1-IN))
      ((Member Final-Accum-1 :x) Fill-1-OUT))
F-212 (For-all (:x)
      ((Member Final-Accum-1 :x) Fill-1-OUT)
      (Or (And ((Member Rest-1 :x) Fill-1-IN)
                ((Member Rest-2 :x) Fill-1-IN))
          ((Member C-1 :x) Fill-1-IN)))

```

This output now flows to the join JOIN-3 which has other unavailable inputs. REASON, therefore, returns to the next case of the test $\rightarrow?$. In this case, it assumes the negation of the unprovable clause from CASE-1 and then attempts to prove that the *when* clause of CASE-2 holds. Thus, REASON assumes that the uid's of the two objects FIRST-1 and FIRST-2 are distinct. Since the uid is a one-to-one property this indicates that FIRST-1 and FIRST-2 are distinct. Furthermore, since the uid is a number and since REASON knows that these two numbers are distinct, it asserts that one of the numbers must be larger than the other. The following assertions result:

```

F-299 (Select Case-2  $\rightarrow?$ )
F-98 ((Not (Equal Number-1 Number-2))  $\rightarrow?$ -IN) (Case-split-assumption (F-299) (F-99))
F-301 (Not (Id First-1 First-2)) (UID-Not- $\rightarrow$  F-98 F-42 F-43)
F-302 (Or ((< Number-1 Number-2)  $\rightarrow?$ -IN) (NumProp F-98)
      ((< Number-2 Number-1)  $\rightarrow?$ -IN))

```

REASON now attempts to prove the *when* clause of CASE-2, however, this too can be seen to be unprovable. It then sets up the next stage of the case-split:

```

F-303 (Select Case-3  $\rightarrow?$ )
F-304 ((Not (< Number-1 Number-2))  $\rightarrow?$ -IN) (Case-Split-Assumption (F-303) (F-99 F-299))
F-305 ((< Number-1 Number-2)  $\rightarrow?$ -IN) (Case-Split-Assumption (F-299)(F-304 F-99))
F-98 ((Not (Equal Number-1 Number-2))  $\rightarrow?$ -IN) (Case-split-assumption (F-303)(F-99 F-299))

```

130 An Example of Symbolic Interpretation

Notice that the justifications are so set up that (1) if either CASE-2 OR CASE-3 is selected the assertion F-98 will be *in*; (2) if CASE-2 is selected F-305 will be *in*, unless there is some reason found to believe its negation; (3) If CASE-3 is selected F-304 and F-98 will be *in*.

The assertion F-305 makes CASE-2 applicable. REASON asserts that CASE-2 is applicable and since there are no output assertions to add, it follows the conditional-control-flow link from CASE-2 to the segment RW-2 which takes the cdr of its input LIST-1 and then calls F11 recursively. Notice that the pre-requisite conditions of F11 are met trivially in this case. The cdr of LIST-1 must be a sorted list as noted earlier; the second input is LIST-2 which is known to be sorted; finally, the CURRENT-ACCUMULAND input is CA which was known to be sorted in reverse order. Thus, F11 is applicable within this sub-plan as well. The output assertions of this application of F11 which REASON names F11-2 are similar to those above:

```
F-310 ((Object-type Final-Accum-2 Reverse-Sorted-List) F12-1-Out)
F-311 (For-all (:x)
      ((Member CA :x) F11-1-IN)
      ((Member Final-Accum-2 :x) F11-1-Out))
F-312 (For-all (:x)
      (And ((Member Rest-1 :x) F12-2-IN)
            ((Member List-2 :x) F12-2-IN))
      ((Member Final-Accum-2 :x) F12-2-OUT))
F-313 (For-all (:x)
      ((Member Final-Accum-2 :x) F12-2-OUT)
      (Or (and ((Member Rest-1 :x) F12-2-IN)
                ((Member List-2 :x) F12-2-IN))
          ((Member CA :x) F12-2-IN)))
```

The justification of these assertions which I have omitted for brevity points back to the relevant spec clause, output object, and applicability assertions. The output of this segment leads to the join JOIN-2 which is waiting for another case's input. REASON now turns to the final case of $\rightarrow 7$. As shown above, REASON assumes in this case that it is false that NUMBER-1 is smaller than NUMBER-2. It then concludes by disjunction elimination that NUMBER-2 is smaller than NUMBER-1:

```
F-314 (((< Number-1 Number-2) <?-OUT) (Disj-Elim F-304 F-302))
```


Actually, I have been taking a slight liberty in the justifications I have shown since as each assertion is moved along a flow link, a new assertion is created with a new situation tag. I have used the fact name of the original assertion in these justifications as a notational convenience.

In any event F-314 is all that is needed to conclude that the third case of the test is applicable. Control therefore flows to RW3 which produces assertions similar to those of RW2. The control-flow link from RW3 to JOIN-2 is now active.

JOIN-2 produces a single output object which is the join of the output produced by RW1, RW2, and RW3. These are the final accumulations produced by the internal recursive calls to F11. REASON names this output of JOIN-2 FA. This output then flows to JOIN-3 where it is joined with the output of JOIN-1. Examination of the diagram shows that the output of JOIN-1 is CURRENT-ACC, the input to the outer F11, since CURRENT-ACC flows to both cases of the join. Thus, the two inputs to JOIN-3 are FA and CURRENT-ACC; REASON names the output of this join FINAL-ACCUM-0. The plan diagram specifies that this is the output of the main segment F11. Symbolic interpretation is, therefore, complete and REASON now tries to prove the asserts of the main segment.

There are three things to be proved: (1) All elements of the INTERSECTION are accumulated (2) Nothing is lost from the CURRENT-ACCUMULATION input (3) Nothing extraneous is accumulated. I will show the proof of first of these claims. This is stated formally as follows:

```
(For-all (:x)
  (And ((Member List-1 :x) F11-IN)
        ((Member List-2 :x) F11-IN))
        ((Member Final-Accum-0 :x) F11-OUT)))
```

To begin the proof of this statement REASON creates an anonymous object to stand for the variable of the quantified statement and then assumes the antecedent clause with this anonymous object substituted for the variable.

```
F-1000 (And ((Member List-1 Obj-1) F11-IN)
              ((Member List-2 Obj-2) F11-IN))
```

132 An Example of Symbolic Interpretation

REASON also establishes the sub-goal for the quantified statement:

```
F-90 (Goal ((Member Final-Accum-0 Obj-1) Fill-OUT) (Achieve-goal ...))
      for ((For-all (:x)
              (And ((Member List-1 :x) Fill-IN)
                    ((Member List-2 :x) Fill-IN))
              ((Member Final-Accum-0 :x) Fill-OUT))
      in ((And ((Member List-1 Obj-1) Fill-IN)
              ((Member List-2 Obj-2) Fill-IN))))
```

The antecedent of the quantified statement is then expanded, yielding the two conjuncts:

```
F-1001 ((Member List-1 Obj-1) Fill-IN)
F-1002 ((Member List-2 Obj-2) Fill-IN)
```

however, rules relating to list structure conclude from these that both lists are not empty.

```
F-1003 ((Not (Object-type List-1 Empty-List) Fill-IN)) (List-def F-1001)
F-1004 ((Not (Object-type List-2 Empty-List) Fill-IN)) (List-def F-1002)
```

This brings in the applicable assertions for the non-null cases of the two NULL? tests, which in turn causes the conditional-control-flow link from the NULL? test to REAL-WORK to be declared active. This in turn brings in the assertion saying that CASE-1 of JOIN-3 is applicable; the output of JOIN-3 is therefore now declared to be 10 to the output of REAL-WORK which is FA. This triggers the identification mechanisms to create a new subgoal in which FINAL-ACCUM-0 is replaced by FA.

```
F-91 (Goal ((Member FA Obj-1) Fill-OUT)
      for ((For-all (:x)
              (And ((Member List-1 :x) Fill-IN)
                    ((Member List-2 :x) Fill-IN))
              ((Member FA :x) Fill-OUT)) (Achieve-goal 1))
      in ((And ((Member List-1 Obj-1) Fill-IN)
              ((Member List-2 Obj-2) Fill-IN))))
```

The data-base is now quiescent. REASON next expands the antecedents of the quantified statement.

F-1005 (Or ((First List-1 Obj-1) F11-IN) (F-1001 List-Mem-definition)
 ((Member [Rest List-1] Obj-1) F11-IN))

F-1006 (Or ((First List-2 Obj-1) F11-IN) (F-1002 List-Mem-definition)
 ((Member [Rest List-2] Obj-1) F11-IN))

The reference expressions in both expressions can be resolved since F-207 and F-208 state what the REST of each list is. Notice that although these facts are tagged with situation tag CDR-1-OUT, there are no side-effects in this program and all assertions except those involving newly created objects may be moved back through any segment to the beginning of the program. (REASON has a different fact name for the same facts in the initial situation, however, for simplicity of presentation I am ignoring this detail). We obtain:

F-1007 (Or ((First List-1 Obj-1) F11-IN) (Reference-Resolution F-1005 F-207)
 ((Member Rest-1 Obj-1) F11-IN))

F-1008 (Or ((First List-2 Obj-1) F11-IN) (Reference-Resolution F-1006 F-208)
 ((Member Rest-2 Obj-1) F11-IN))

There are now several strategies which REASON might pursue. It could attempt a proof by cases, splitting either of the above disjunctions (F-1007 or F-1008) or the disjunction of select statements associated with either the test =() or the join JOIN-2. The current version of REASON, has a preference for splitting disjunctions arising from the goal, rather than case-split or join oriented disjunctions. REASON attempts to show the goal F-91 by splitting F-1007. It first assumes:

F-1009 ((First List-1 Obj-1) F11-IN) (Case-split-assumption F-1007 ...)

However, the system has already asserted F-40 which states that the first of List-1 is First-1. Thus, an identification is made, leading to the following:

F-1010 (Id First-1 Obj-1) (Parts-Id F-1009 F-40)
 F-1011 ((Unique-Id Obj-1 Number-1) Maknum-1-Out) (Identification F-1010 F-42)

Now the system chooses to case-split the second disjunction F-1008. It obtains

134 An Example of Symbolic Interpretation

F-1012 ((First List-2 Obj-1) Fill-IN) (Case-split-assumption F-1000 ...)
 F-1013 (Id First-2 Obj-1) (Parts-Id F-1012 F-41)
 F-1014 ((Unique-Id Obj-1 Number-2) Maknum-1-Out) (Identification F-1013 F-44)

This, in turn creates another identification:

F-1015 (Id Number-1 Number-2) (Func-Prop-Id F-1011 F-1014)

However, F-1015 means that CASE-1 of the test -<>? is applicable. Thus, the assertions pertaining to RV-1 come *in* since this is the segment which follows from case-1. Further identification follows:

F-1016 ((Member C-1 Obj-1) CONS-1-OUT) (Identification F-206 F-1010)

This last assertion interacts with the quantified statement F-211 to create the new assertion:

F-1017 ((Member Final-Accum-1 Obj-1) Fill-1-Out) (For-All F-211 F-1016)

Finally, since the control-flow coming into CASE-1 of JOIN-2 is active, the assertion stating the applicability of this case comes *in*. This, in turn, brings *in* an ID assertion stating that FINAL-ACCUM-1 is identical to the output of JOIN-2 which is FA. This triggers another round of identifications:

F-1018 (Id Final-Accum-1 FA) (Join-Select ...)
 F-1019 ((Member FA Obj-1) Fill-1-Out) (Identification F-1017 F-1018)

This assertion then passes directly through JOIN-2 and JOIN-3 satisfying the desired goal. However, this was only the first case of the second case-split. REASON now revokes the assumption F-1012 and makes the assumption that OBJ-1 is a member of REST-2, the CDR of List-2.

F-1020 ((Member Rest-2 Obj-1) Fill-IN) (Case-split-assumption F-1000 ...)

Notice that all the identifications triggered by the assumption F-1012 are now *out*, since that assumption has been *outed* by the proof-by-cases mechanism. However, REASON still has the assertion that OBJ-1 is identical to FIRST-1 since it has not yet revoked that assumption. Rules relating to list structure trigger, this time concluding that the UNIQUE-ID of OBJ-1 is greater than that of FIRST-2 since it is a member of the CDR of a sorted list of which FIRST-2 is the CAR.

F-1021 ((> [Unique-Id Obj-1] [Unique-Id First-2]) F11-IN) (Lists F-1020 F-200 F-41 F-20)

The reference expressions in the above assertion need to be resolved and both referents are available. We get:

F-1022 ((> Number-1 Number-2) F11-IN) (Ref-Resolution F-1021 F-1011 F-43)

Rules reflecting knowledge about numbers and the one-to-one character of the *uid* now trigger:

F-1023 ((Not (< Number-1 Number-2)) F11-IN) (Num-Prop F-1022)

F-1024 ((Not (Equal Number-2 Number-1)) F11-IN) (Num-Prop F-1022)

F-1025 (Not (Id First-2 Obj-1)) (UID-Prop F-1024 F-1011 F-43)

Assertions F-1025 and F-1023, however, imply that CASE-1 and CASE-2 of the test *-(<?)* are inapplicable and F-1023 implies that CASE-3 of the test is applicable. Therefore, *rw3* is the segment to which control is transferred. However, during the symbolic evaluation of *rw3*, a quantified statement was created which stated that any object which was both a member of *LIST-1* and *REST-2* is a member of *FINAL-ACCUM-3*, the output of the internal call to *F11*. We obtain:

F-1026 ((Member Final-Accum-1 Obj-1) *RW3-OUT*) (For-All F-1020 F-1009 ...)

As above, this passes through the joins directly and the desired goal is achieved in this case. Thus, REASON has finished proving that if the object *obj-1* is the first object of *LIST-1* it will be a member of the final output. REASON now considers the other case of the disjunction F-1007. It assumes:

F-1027 ((Member Rest-1 Obj-1) F11-IN) (Case-Split-Assumption F-1007 ...)

which triggers a set of deductions similar to those which followed from the assumption F-1020. REASON again decides to try proof by cases on the disjunction F-1008. The first case of this proof brings back in the assumption F-1012 which states that the *FIRST* of *LIST-2* is *obj-1*. This selects CASE-2 of the test *-(<?)* and rules out the others as above. The quantified statement produced by symbolic evaluation of the internal call to *F11* within *rw2* is triggered just as above, leading to the desired conclusion. This leaves only one final case to consider. REASON brings in the assumption F-1020, stating that *obj-1* is a member of the *CDR* of *LIST-2*.

136 An Example of Symbolic Interpretation

REASON now has the following four facts *in*

F-1001 ((Member List-1 Obj-1) F11-IN)
F-1002 ((Member List-2 Obj-2) F11-IN)
F-1020 ((Member Rest-2 Obj-1) F11-IN)
F-1027 ((Member Rest-1 Obj-1) F11-IN)

The desired goal has not yet been obtained, so REASON finally resorts to splitting the disjunction associated with JOIN-2. First it assumes that CASE-1 of the join is selected, *in*ing the assumption that CASE-1 of the test <C> is selected. Thus RV1 is active. However, associated with its internal call to F11 is a statement that any member of both REST-1 and REST-2 will be a member of the output. This will satisfy the sub-goal as we've seen above. Selecting CASE-2 of JOIN-2 will take control to RV2 which says that any object which is in REST-1 and LIST-2 will be a member of the output. Finally, Selecting CASE-3 will lead to RV3 and the requirement that the object be in LIST-1 and REST-2. Since OBJ-2 satisfies all of these requirements it is a member of the output of JOIN-2 in all cases. Thus, the proof is complete.

The proofs of the other goals follow along similar lines, involving no mechanisms other than those shown so far. The proof shown above was constructed by the first implementation of REASON, although some technical details were different. In the next two chapters I will turn to the issues of categorizing standard plan fragments which motivated the new implementation effort.

Chapter 8: The Temporal Viewpoint

The problem with the proof given in the last chapter is that it involved a lot of hard work proving things which most programmers would recognize as examples of things they already know. This puts a premium on the recognition of pre-proven "plan fragments". Most previous research on the use of pre-proven schemata such as that of [Gerhart, 1975] has relied on syntactic templates and correctness preserving transformations on the program syntax. In contrast, the Programmer's Apprentice represent its knowledge of standard programming techniques in the plan formalism, using data-flow and control-flow links to abstract away from the syntax of the programming language. In addition, we use symbolic evaluation and a situational logic to talk about the internal states of the computation.

Two distinct segmentations of a program can be made, each revealing different aspects of its teleological structure. The first is a segmentation of the surface features, called the *surface viewpoint* which abstracts out the communication primitives of the programming language; the second expands the program into a sequence of situations which are regrouped into the *temporal viewpoint*. This technique allows programs to be described and catalogued in a high-level vocabulary which is suitable for use as a very-high-level programming language or as a command language to a programmer's apprentice system.

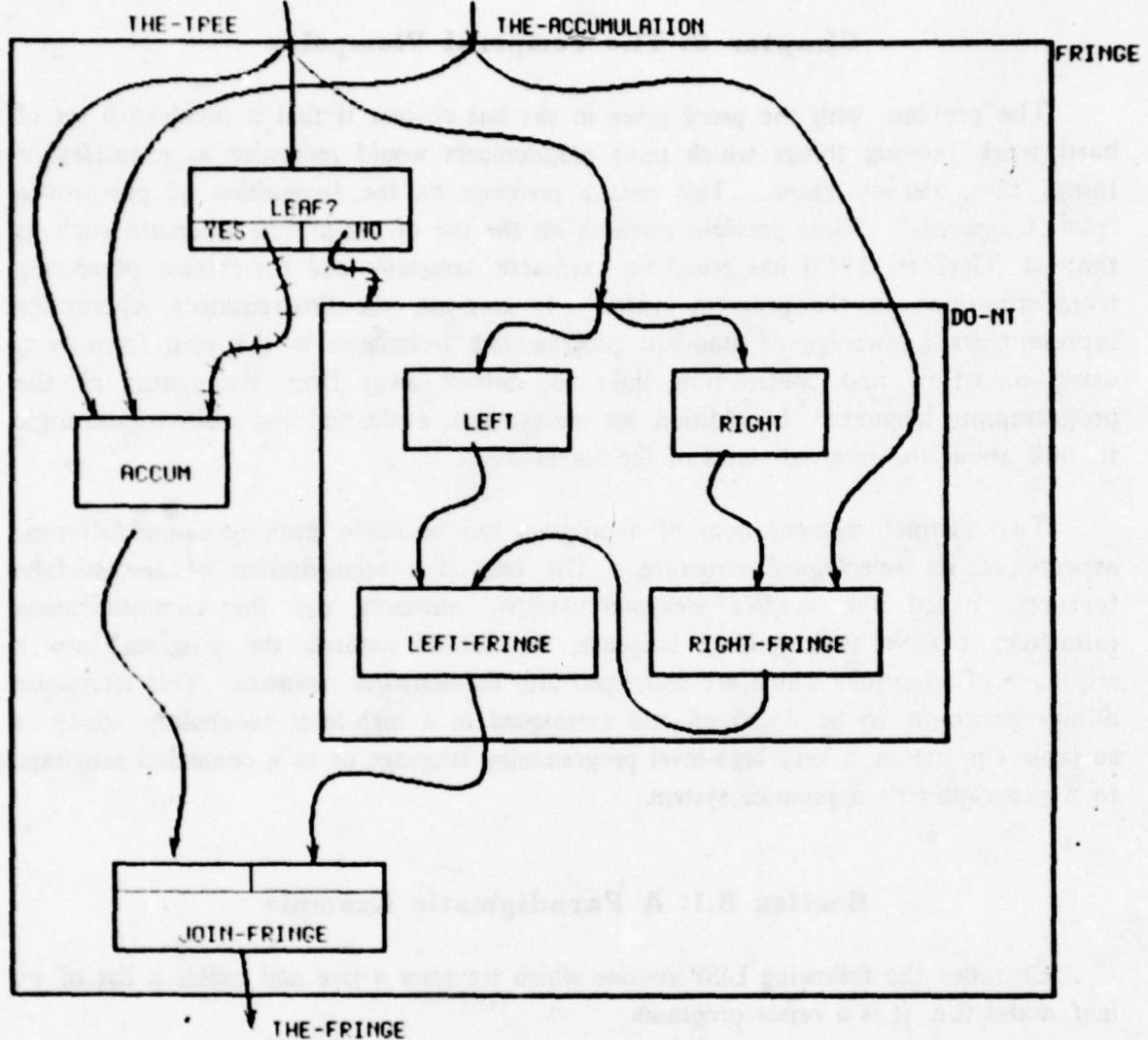
Section 8.1: A Paradigmatic Example

Consider the following LISP routine which traverses a tree and builds a list of its leaf nodes (i.e. it is a FRINGE program):

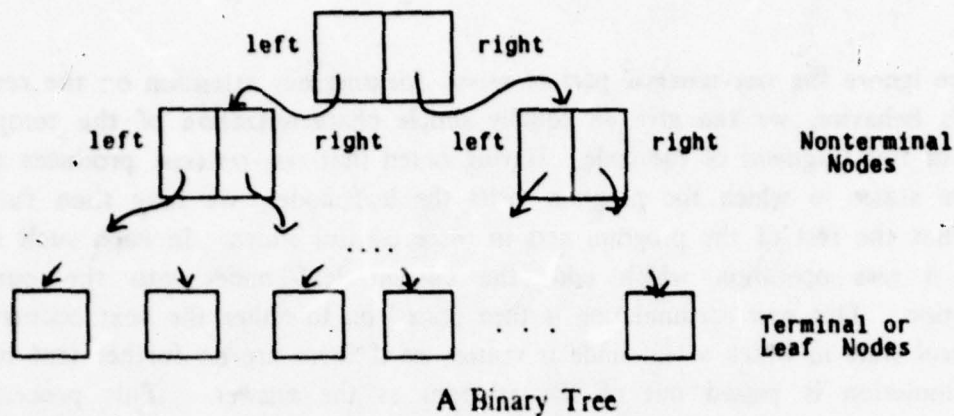
```
(define fringe (tree)(fringe-1 tree nil))

(define fringe-1 (current-node accumulation)
  (cond
    ((test-leaf current-node)(cons current-node accumulation))
    (t (fringe-1 (left current-node)
                  (fringe-1 (right current-node)
                            accumulation))))))
```

For Complex Program Understanding



The program might be paraphrased as follows: If the **CURRENT-NODE** is a leaf node then cons it onto **ACCUMULATION**, the current list of leaf nodes; if it is not then add to **ACCUMULATION** all those leaf nodes which are daughters of the right node of the **CURRENT-NODE**. Then add to the result of that computation all those leaf nodes which are daughters of the left node of the **CURRENT-NODE**. If started with the **TREE** as the **CURRENT-NODE** and **NIL** as the accumulation, the program will build a list containing exactly the leaf nodes of **TREE**.



A standard proof of correctness of the above program would closely follow the above description, using induction to argue that the right recursive call accumulates all the leaf nodes of the right branch and then that the left recursive call accumulates the others. However, such a proof does not make use of knowledge which is second hand to most LISP programmers. The `FRINGE` program follows a standard pattern of double recursion on the branching structure of the tree. In this case the standard tree-recursion is augmented by the presence of (1) A `CONS` and (2) A second argument in the function definition of `FRINGE-1`; the purpose of this argument is to accumulate the set of leaf nodes. Were we to ignore these extra features, we would be left with a program which does nothing but traverse a tree.

```
(define traverse-a-tree (tree)
  (cond ((test-leaf tree)
        (t (traverse-a-tree (left tree))
            (traverse-a-tree (right tree)))))
```

Although the logic underlying this code is a cliché of LISP programming, it is not possible to specify this segment's behavior using standard input/output descriptions. Indeed, `TRAVERSE-A-TREE` produces no outputs at all, and thus, has no I/O behavior to specify. However, the segment does have useful temporal behavior: during its computation it visits every node of the input tree. Secondly, during its computation `TRAVERSE-A-TREE` filters the nodes of the tree into leaf nodes and non-leaf nodes, creating a set of control states in which precisely the leaf nodes are available. Thus, the I/O descriptions of Hoare [Hoare, 1969] logic are inadequate for this purposes and a logic of greater strength is needed. Those interested in the logic of computer programming are now studying such logics [Pnueli, 1977] [Pratt, 1978] although with other purposes

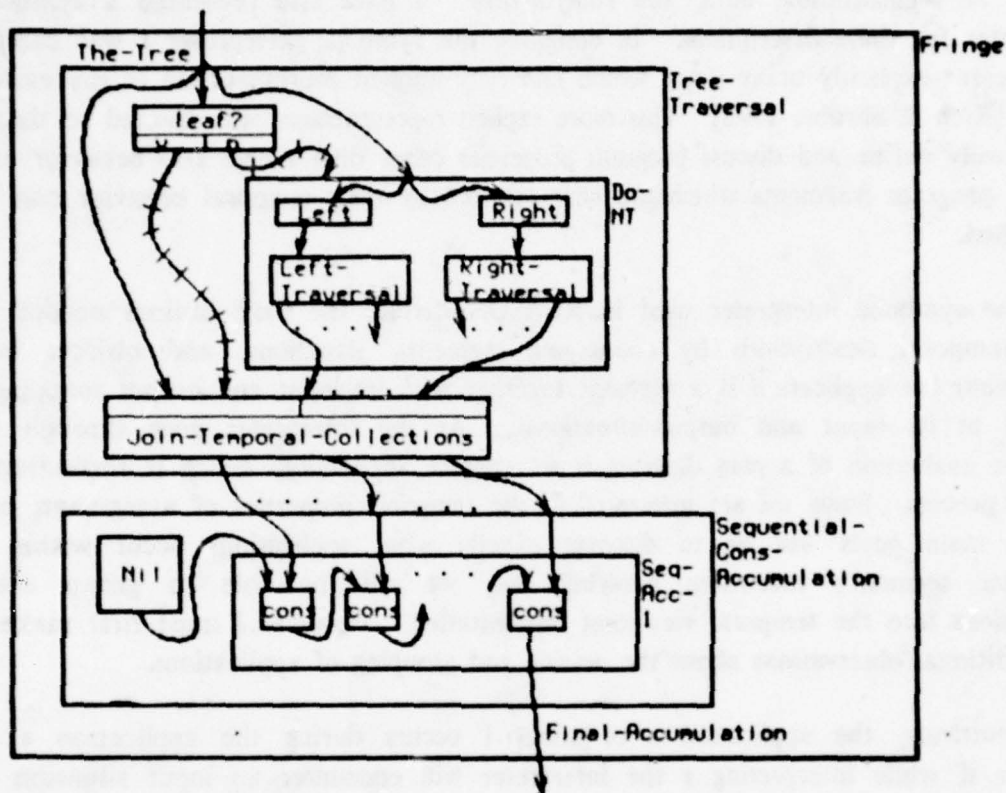
For Complex Program Understanding

in mind.

If we ignore the `TREE-TRAVERSAL` part of `FRINGE`, focusing our attention on the rest of the code's behavior, we can give an equally simple characterization of the temporal behavior of this fragment of the code. Having noted that `TREE-TRAVERSAL` produces a set of control states in which the program visits the leaf nodes, we may then further observe that the rest of the program acts in these control states. In each such state there is a `cons` operation which adds the current leaf node onto the current accumulation. This new accumulation is then passed on to either the next occurrence of a control state in which a leaf node is visited, or if there are no further leaf nodes the accumulation is passed out of the program as the answer. This process of sequentially accumulating additional values is also a cliché of LISP programming which I will refer to as *sequential-cons-accumulation*.

This leads to a different view of the `FRINGE` program. We may now regard it as a "composition" (in the sense of functional composition) of a `TREE-TRAVERSAL`, a `LEAF-FILTER`, and a `SEQUENTIAL-CONS-ACCUMULATION`. In contrast to normal compositions which communicate by passing a set of data-objects each of which exists as a unified object at the time of the functional invocation, this composition instead passes *temporal collections* of values which can be regarded as a unified object only by abstracting away from the program's sequential behavior. This view of a recursive program as being composed of a generator together with a consumer is used in the languages CLU [Liskov et. al., 1977] and ALPHARD [Wulf et. al., 1976] and is the basis of both the language APL [Iverson, 1962] and the loop analyzer used in the programmer's apprentice project [Waters, 1978].

It would be desirable to be able to construct the following (more natural) proof of the fringe program: First, we already know that `TREE-TRAVERSAL` visits every node of the tree and that it filters out all but the leaf nodes. Second, each such node is passed to `SEQUENTIAL-CONS-ACCUMULATION`. Third, we already know that `SEQUENTIAL-CONS-ACCUMULATION` will return a list of exactly those objects which it was passed as inputs. Finally, since the output of `SEQUENTIAL-CONS-ACCUMULATION` is the output of `FRINGE`, it follows that the `FRINGE` program produces a list of exactly the leaf nodes of its input tree.



Temporal View of Fringe Program
Tree Traversal and Sequential-Cons-Accumulation

The advantage of this method is that we can make use of previously constructed proofs of the properties of TREE-TRAVERSAL and SEQUENTIAL-CONS-ACCUMULATION; however, in order to reap this advantage we will have to construct rules of inference which allow us to prove temporal properties of program segments and which tell us when it is allowable to apply these properties.

For Complex Program Understanding

In previous chapters a formalism was developed for describing programs while abstracting away from the primitives of the programming language, using instead the notions of segmentation, data, and control-flow. I have also presented a symbolic interpreter for these descriptions. In designing this symbolic interpreter I was careful to represent explicitly many items which had only implicit representation in my earlier system [Rich & Shrobe, 1976]. This more explicit representation was created so that I could easily define and discuss program properties other than simple I/O behavior and so that program fragments which are characterized by their temporal behavior can be catalogued.

The symbolic interpreter used in REASON defines the basic notions needed to build temporal descriptions by connecting segments, situations, and objects into *applications* (an application is a segment together with its input and output mappings, as well as its input and output situations). As the interpreter goes through its symbolic evaluation of a plan diagram it records the applications which it encountered in that process. Since we are interested in the temporal properties of a segment, one of our main goals will be to discover exactly what applications occur within a particular segment's execution; knowing this, we will be able to group these applications into the temporal viewpoint segmentation. However I must first make a few additional observations about the nesting and grouping of applications.

Intuitively, the application *A* of *SEGMENT-1* occurs during the application *a* of *SEGMENT-2* if while interpreting *a* the interpreter will encounter an input situation in which it applies *SEGMENT-1* to some set of inputs. *SEGMENT-1* together with the set of inputs, the set of outputs, the mappings of these to the input and output names of *SEGMENT-1*, and the input and output situations constitute the application *A*.

More formally, we say that *an application A of segment-1 occurs within an application B of segment-2* if the following conditions hold:

- (a) The segment of application *A* is *SEGMENT-1* and the segment of application *a* is *SEGMENT-2*.
- (b) *SEGMENT-1* is a sub-segment of *SEGMENT-2*.
- (c) Each data-flow link terminating at *SEGMENT-1* originates either at an input of *SEGMENT-2* or at an output of some other application *c* where *c* occurs within *a*.
- (d) Each data-flow link originating at *SEGMENT-1* terminates at either an output of *SEGMENT-2* or at an input of some other application *c* where *c* occurs within *a*.
- (e) Each conditional-control-flow link which terminates at *A* originates at an

application *c* which occurs within *a*. Furthermore, the conditional-control-flow link originates at an *applicable* case of the segment of *c*.

The relationship is transitive, so that if *A* is an application within *a* and *a* is an application within *c*, then *A* is an application within *c* as well. We express this as follows:

(application-within A B)

which states that *A* is an application within *a*.

Frequently it is more useful to talk about applications of segments of a particular (plan or spec) type rather than applications of a particular segment. We use the following notation to express this idea:

(application-of-type <plan-type> <application>)

The above can be defined by the following equivalence:

(application-of-type type-1 appl-1)

=

(plan-type [segment-part appl-1] type-1)

We noted earlier that in the TREE-TRAVERSAL plan there is an application of a segment of type TREE-TRAVERSAL for every node of the tree. However, some of these applications occur within the sub-segment called on the left branch while others involve the right. In this case, we are concerned not only with applications of the particular surface sub-segments, but also with applications of all segments of type tree traversal.

We say that there is an *occurrence of plan-type type-1 within the application A of segment-1* if:

- (i) There is an application *a* of segment SEG-1 within the application *A*
- (ii) The plan-type of SEG-1 is TYPE-1.

We express this as follows:

For Complex Program Understanding

(occurrence-within <application-A> <plan-type> <application-B>)

the following equivalence holds:

```
(occurrence-within appl-1 type-1 appl-2)
=
(and (application-within appl-1 appl-2)
     (application-of-type type-1 appl-2))
```

We can now state a formal property of the TREE-TRAVERSAL fragment which will be useful throughout the rest of this discussion, namely that it visits every node:

```
(for-all (:appl-1) (application-of-type tree-traversal :appl-1)
  (for-all (:node) (node [input appl-1 the-tree] :node)
    (there-is (:appl-2) (occurrence-within appl-1 tree-traversal :appl-2)
      such-that (input :appl-2 the-tree :node))))
```

It is convenient to think of sets of applications actually being aggregated into a segment. This is done using the notion of an *occurrence set*. The *occurrence set of plan-type type-1 within the application A* is the set of all applications within A whose plan-type is type-1. Intuitively, the occurrence set of plan-type type-1 is a virtual segment consisting of that part of the program's temporal history which includes applications of a particular type. In an FRINGE program, for example, the occurrence set of type FRINGE is essentially the tree traversal fragment of the program, consisting of all the recursive calls to FRINGE.

Section 8.2: Situations and Orderings

Several rules are used to impose a partial order (representing temporal ordering) on the situations occurring in a plan diagram. Most obvious are those imposed by the relationship of the situations to the segments of the plan-diagram. The following rules capture the constraints that (1) The input situation of a segment precedes its output situation, (2) The input situation of a main segment precedes the input situation of any of its sub-segments, (3) The output situation of any sub-segment precedes the output situation of its main segment.

```

(Rule ((:f (Is-a :segment Segment))
      (:g (Input-situation :segment :in-sit))
      (:h (Output-situation :segment :out-sit)))
  (Assert (Comes-before :in-sit :out-sit)
    (Seg-sit-order :f :g :h)))

(Rule ((:f (Sub-segment :main-seg :sub-seg))
      (:g1 (Input-situation :main-seg :in-sit-1))
      (:g2 (Input-situation :sub-seg :in-sit-2))
      (:h1 (Output-situation :main-seg :out-sit-1))
      (:h2 (Output-situation :sub-seg :out-sit-2)))
  (Assert (Comes-before :in-sit-1 :in-sit-2)
    (Nested-seg-order :f :g1 :g2))
  (Assert (Comes-before :out-sit-2 :out-sit-1)
    (Nested-seg-order :f :h1 :h2)))

```

Data-flow and control-flow links are the only other constraint ordering the situations. If there is a data- or control-flow link between two situations then the output situation of the first segment must precede the input situation of the other.

```

(Rule ((:f (Dataflow :seg-1 :seg-2))
      (:g (Output-situation :seg-1 :out-sit))
      (:h (Input-situation :seg-2 :in-sit)))
  (Assert (Comes-before :out-sit :in-sit)
    (dflow-order :f :g :h)))

(Rule ((:f (Controlflow :seg-1 :seg-2))
      (:g (Output-situation :seg-1 :out-sit))
      (:h (Input-situation :seg-2 :in-sit)))
  (Assert (Comes-before :out-sit :in-sit)
    (dflow-order :f :g :h)))

(Rule ((:f (Conditional-Control-flow (:seg-1 :case-1) :seg-2))
      (:g (Output-situation :seg-1 :out-sit))
      (:h (Input-situation :seg-2 :in-sit)))
  (Assert (Comes-before :out-sit :in-sit)
    (dflow-order :f :g :h)))

```


Since these constraints in general only impose a partial ordering on the situations, plan diagrams may be thought of as representing a (pseudo) parallel computation which imposes only the minimal constraints on segment ordering necessary to achieve the goals of the segment specified by the plan-diagram.

Finally we may define a notion of a situation belonging to a particular application A , namely that it is an input or output situation of some application occurring within A :

```
(situation-of-application A sit)
#
(There-is (:b) (application-within A :b)
 such-that (or (input-situation :b sit)
               (output-situation :b sit)))
```

Section 8.3: Temporal Collections

I would like to define the idea of a collection of objects distributed in time, rather than gathered together in a data-structure. The motivation of this, as mentioned earlier, is to be able to describe what a program fragment like *TREE-TRAVERSAL* does. I will show that *TREE-TRAVERSAL* may be regarded as producing such a temporal collection whose members are exactly the nodes of the input tree.

A *temporal collection* is a set of pairs of objects and situations such that each object exists in the situation with which it is paired. We may talk about elements of the collection and their object and situation parts as follows:

```
(Element <temporal-collection> <element>)
(Object-part <element> <object>)
(Situation-part <element> <situation>)
```

Since the second two of the predicates are functions, they may be referred to with the bracket notation.

C is a temporal collection of the application *A* if:

- (i) *c* is a temporal collection
- (ii) Every element of *c* has a situation part which is a situation of application *A*.

It is easy to create temporal collections by picking applications of sub-segments within some main segment. For each such sub-segment we can choose an (input) output object and pair it with the (input) output situation. However, it is usually more useful to consider an occurrence set of a particular type and to construct the set formed by those objects which are assigned to a particular input (output) port of each occurrence. These objects are then paired with the input (output) situations of each occurrence to form a temporal collection.

For example, we may consider the occurrence set of type *TREE-TRAVERSAL* within any application of *TREE-TRAVERSAL*. For each such occurrence, choose the input which is assigned to *TREE-TRAVERSAL*'s input port *THE-TREE*. Pair each such object with the input situation of the occurrence to which it is an input. This temporal collection contains exactly the nodes of the tree input to the outermost application of *TREE-TRAVERSAL*.

From now on I will use the term *temporal collection* only to refer to temporal collections generated by the <name> input (output) of the occurrence set of <plan-type> within the application *A*. It is denoted as follows:

(Temporal-Collection *A* The-Tree Tree-Traversal *c*)
 (Temporal-Collection <application> <input-name> <plan-type> <collection-name>)

which says that *c* is the temporal collection generated by *THE-TREE* input port of the occurrence set of type *TREE-TRAVERSAL*.

So far I have talked about a temporal collection as a set of pairs of objects and situations. Such pairs are said to be *elements* of the temporal collection. However, the interest is usually not with the pairs but only with the object parts of the pairs. An object is a *member of a temporal collection* if there is an element of the temporal collection whose object part is the object in question. Notice that under the element relationship the temporal collection is a set; we are only interested in the presence of a pair in the collection, not the number of occurrences. However, under the membership relationship, the temporal collection is a multi-set, with objects occurring more than once.

For Complex Program Understanding

As with other data-structures, it is frequently important to have an ordering relationship on the objects in a temporal collection. The temporal ordering of the situations provides a natural method for defining such an ordering. We say that *element-1 precedes element-2 in the temporal collection C* if:

- (i) There is an element of *c* whose object part is *ELEMENT-1* and whose situation part is *SITUATION-1*.
- (ii) There is an element of *c* whose object part is *ELEMENT-2* and whose situation part is *SITUATION-2*.
- (iii) *SITUATION-1* comes-before *SITUATION-2*.

Notice that since situations are only partially ordered, the elements of a temporal collection are in general only partially ordered. If the ordering of the elements of a temporal collection is total, then we say that the collection is a *temporal sequence*.

Section 8.4: Temporal Collections Inputs and Outputs

We now extend the specification language to allow temporal collections to serve as segment inputs and outputs. Thus, the specs for *TREE-TRAVERSAL* may now be stated as follows:

```
(defspecs tree-traversal
  (Inputs: the-tree)
  (Expect: (Object-type the-tree Binary-tree))
  (Outputs: the-nodes)
  (Assert: (Object-type the-nodes temporal-collection)
    (For-all (:the-node) (node the-tree :the-node)
      (member the-nodes :the-node))
    (For-all (:the-node) (member the-nodes :the-node)
      (node the-tree :the-node))))
```

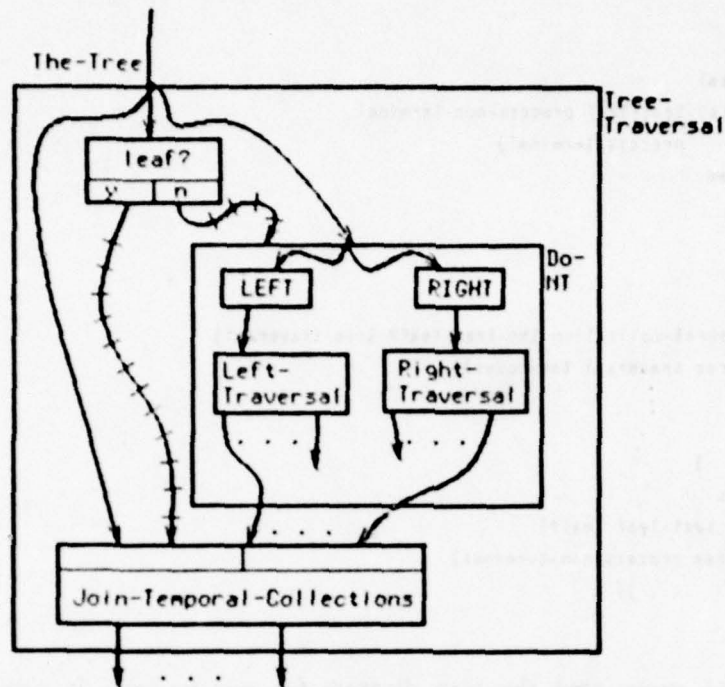
Which states that *TREE-TRAVERSAL* produces a temporal collection output whose elements are exactly the nodes of the input tree. We can now state the plan diagram assertions which link the temporal collection generated by the occurrence set of *TREE-TRAVERSAL* to the output *THE-NODES*.


```

(defplan tree-traversal
  (sub-segments: test-leaf process-non-terminal
                 process-terminal)
  (flow-diagram:
    .
    .
    .
    (dataflow
      (temporal-collection the-tree leaf? tree-traversal)
      (output tree-traversal the-nodes))
    .
    .
    )
  (constraints:
    (spec-type test-leaf leaf?)
    (plan-diagram process-non-terminal)
    ... ))

```

It is now possible to prove that the plan diagram for TREE-TRAVERSAL is consistent with its specs. In doing so REASON uses a form of computational induction, assuming that a temporal property holds of all occurrences of a particular plan type within the main application. If on the basis of this assumption it can deduce that the same property holds for the main application, then it is legitimate to conclude that the property is true for any application of that plan-type. (This is actually only weak correctness in the Hoare sense since this does not prove termination). I call this form of computational induction *plan-type computational induction*.



Plan Diagram of Tree Traversal

To begin the proof REASON assumes that there is an arbitrary application **TREE-TRAVERSAL-1** of a segment of plan-type **TREE-TRAVERSAL**. Anonymous objects are chosen to stand for the input situation of the application. Anonymous objects are also chosen to stand for the inputs to the application.

```
(application-of-type tree-traversal tree-traversal-1)
(input tree-traversal-1 the-tree tree-1)
(input-situation tree-traversal-1 sit-1)
```

Also the input expectations of the **TREE-TRAVERSAL** specs are asserted in the input situation of the application.

```
((Binary-tree tree-1) sit-1)
```

The data-flow links in the plan diagram show that `TEST-LEAF` is ready for application. `REASON` decides that it is impossible to prove either that `TREE-1` is a terminal or that it is not. Thus, a case-split is created, assuming in the first case that `TREE-1` is a terminal. The plan diagram indicates that no other segments are applicable and that control flows to the main segment's output. The system must prove that the main segment's assert clauses hold for the delivered output objects.

There is only assert clause and it states that every node of `TREE-1` must be present in the output `THE-NODES` which is the temporal collection generated by the occurrence set of `TEST-LEAF`. This is trivially true since the only node of a terminal node is itself, and the only occurrence of `TEST-LEAF` had `TREE-1` as its input. Thus, the plan is valid under the assumption that `TREE-1` is a terminal.

If `TREE-1` is non-terminal then `CASE-2` of `TEST-LEAF` is applicable and `PROCESS-NON-TERMINAL` is ready for application. `REASON` moves through the plan diagram evaluating the sub-segments in turn. Since `REASON` has assumed that `TREE-1` is non-terminal, the segments `LEFT` and `RIGHT` are applicable; objects `LEFT-1` and `RIGHT-1` are created to represent their outputs and assertions are added to represent the fact that `LEFT-1` is the left node of `TREE-1` and that `RIGHT-1` is the right node.

```
(output the-left-node left left-1)
(output the-right-node right right-1)
(left-node tree-1 left-1)
(right-node tree-1 right-1)
```

Data-flow links then map these objects to `LEFT-TRAVERSAL` and `RIGHT-TRAVERSAL` which are applicable since they only require their input to be a binary-tree node. Both `LEFT-1` and `RIGHT-1` are binary-tree nodes since they are nodes of the binary tree `TREE-1`.

`REASON` next attempts to show that the temporal collection generated by the occurrence set of type `LEAF?` includes exactly the nodes of `TREE-1`. The proof is by (plan-type) computational induction. `REASON` assumes that any occurrence `o-1` of plan-type `TREE-TRAVERSAL` within the main application satisfies the property which it wishes to prove, i.e. that the temporal collection generated by the occurrences of type `LEAF?` within `o-1` has exactly the nodes of the input tree as its members. Thus, the occurrences of `LEAF?` within `LEFT-TRAVERSAL` and `RIGHT-TRAVERSAL` each generate a temporal collection including exactly the nodes of `LEFT-1` and `RIGHT-1`.

For Complex Program Understanding

Consider an arbitrary node `NODE-1` of `TREE-1`. By definition `NODE-1` is either `TREE-1` itself or a node of the left node of `TREE-1` or a node of the right node of `TREE-1`. If `NODE-1` is identical to `TREE-1` then it is the input to `TEST-LEAF` which is of type `LEAF?`. If `NODE-1` is not identical to `TREE-1` then it is a node of either `LEFT-1` or `RIGHT-1`. In either of these cases, `REASON` has shown by induction that it is a member of a temporal collection generated by the occurrences of type `LEAF?` within `LEFT-TRAVERSAL` or `RIGHT-TRAVERSAL`. Since these occurrences are within a sub-segment of `TREE-TRAVERSAL`, they are sub-segments of `TREE-TRAVERSAL` itself. Thus all the nodes of `TREE-1` are in the temporal collection generated by the occurrences of `TEST-LEAF.ct`

Section 8.5: Temporal Collection Data-flows

Temporal collections are a useful abstraction mechanism only if they can serve not only as outputs of a segment but also as inputs. In this case, data-flows between segments might involve the flow of a temporal collection output of one segment to a temporal collection input of a second segment. This single data-flow statement is, however, an abstraction of the temporal behavior of the plan, summarizing many identifiable data-flows between sub-segments of the two plans.

For example, in the temporal model of the `FRINGE` program we now have a sub-segment called `TREE-TRAVERSAL` which outputs a temporal collection containing the nodes of its input. This is connected by a data-flow link to `SEQUENTIAL-ACCUMULATION` which takes a temporal collection input. This single link summarizes the fact that each occurrence of type `LEAF?` within `TREE-TRAVERSAL` has a corresponding occurrence of `cons` in `SEQUENTIAL-ACCUMULATION` with a data-flow link connecting the two.

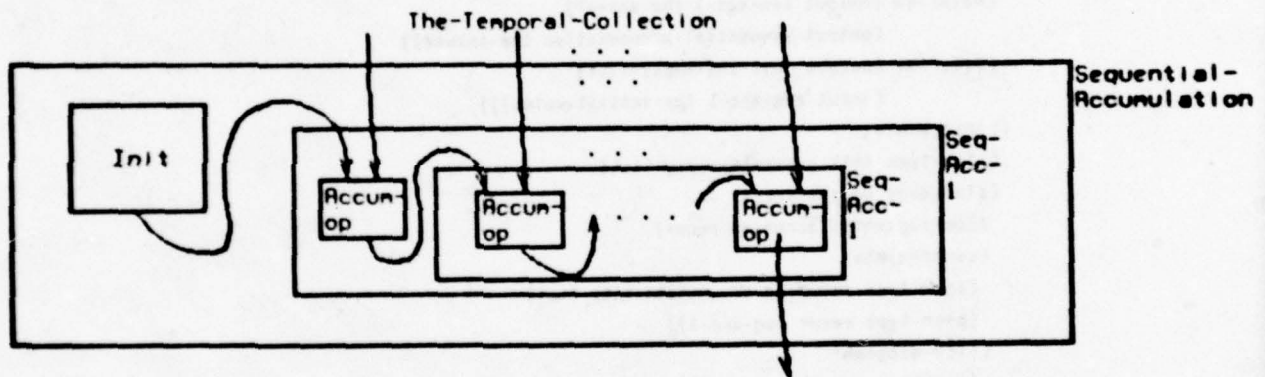
`SEQUENTIAL-ACCUMULATION` can be given a simple set of specs saying that it takes as input a temporal collection and returns as output a single object which contains exactly the same members of the temporal collection.

```

(defspecs Sequential-Accumulation
  (inputs: collection-1)
  (expect: (Object-type collection-1 temporal-collection))
  (output: list-1)
  (assert: (for-all (:member-1) (member collection-1 :member-1)
                    (member list-1 :member-1))
            (for-all (:member-1) (member list-1 :member-1)
                    (member collection-1 :member-1))))

```

The internal plan for SEQUENTIAL-ACCUMULATION is intuitively a cascade of cons segments, each taking one input from the temporal collection and the other input from the output of the preceding cons. The first cons takes NIL as input and the last delivers its output as the output of the whole segment.



Sequential-Accumulation with Temporal Collection Input

In describing the TREE-TRAVERSAL plan fragment I needed the notion of a temporal collection being generated by the occurrence set of the same plan type. The SEQUENTIAL-ACCUMULATION fragment requires an inverse to this idea. We may think of two distinct types of segments, those which generate a collection of values and those which utilize the collection. In the first kind of segment it is convenient to talk about the occurrences of segments of a particular type generating a temporal collection; in the second kind of segment, we require the notion of a temporal collection determining the occurrences of sub-segments of a particular type. When we say that a segment takes a temporal collection as input, we are summarizing the idea that the objects in

154 The Temporal Viewpoint

the temporal collection are in a one to one correspondence with the members of an occurrence set within the segment. We refer to such an occurrence set as the temporal collection generated occurrence set, which we denote as follows

```
(tc-gen-occ <plan-type> <set-of-occurrence-name>)
```

This allows us to represent the flow of objects of a temporal collection to a collection of occurrences by a single data-flow statement as is done in the statements of the following plan diagram (which is shown pictorially above):

```
(defplan sequential-accumulation
  (sub-segments: init seq-acc-1)
  (flow-diagram:
    (dataflow (input sequential-accumulation the-temp-col)
              (input seq-acc-1 the-temp-col))
    (dataflow (output seq-acc-1 the-answer)
              (output sequential-accumulation the-answer))
    (dataflow (output init the-empty-list)
              (input seq-acc-1 the-initial-value)))
  (constraints:
    (plan-type init generate-empty-list)
    (plan-diagram seq-acc-1
      (sub-segments: accum-op recur)
      (constraints:
        (spec-type accum-op accumulate-into-list)
        (plan-type recur seq-acc-1))
      (flow-diagram:
        (dataflow (input seq-acc-1 the-temp-col)
                  (input tc-gen-occ accum-op the-current-value))
        (dataflow (output accum-op new-list)
                  (input recur current-accum))
        (dataflow (output recur the-answer)
                  (output seq-acc-1 the-answer))))))
```

Notice that the data-flow links between the various occurrences of `accum-op` impose a temporal ordering on their execution which (in this case) is total. We require the ordering of elements of the temporal collection to be consistent with the ordering of the segments into which they flow. This is not an issue in the plan for `FRINGE` since the version of `TREE-TRAVERSAL` we are considering is so abstract that it specifies no ordering on the elements generated. Its output and hence the input of `SEQUENTIAL-ACCUMULATION` is totally unordered; any mapping of the elements input to `SEQUENTIAL-`

ACCUMULATION is consistent with the segment ordering of the occurrences of ACCUM-OP.

However, if we were to consider a more specific TREE-TRAVERSAL, say one which traverses in left-to-right, depth-first order, then there would be restrictions in the mapping; given two nodes, the node which is further to the left in the tree, will be mapped into an earlier occurrence of ACCUM-OP and will, therefore, appear earlier in the output of SEQ-ACC-1. This requirement guarantees that the SEQUENTIAL-ACCUMULATION plan will not lose ordering constraints which were of significance to the segment which generated its inputs.

REASON can prove that the SEQUENTIAL-ACCUMULATION plan satisfies its specs quite easily. It once again uses (plan-type) computational induction. In summary, the argument divides into two parts. The simple part is that if SEQ-ACC-1 satisfies its specs of accumulating all the elements of its temporal collection input into the list which is its other input, then SEQUENTIAL-ACCUMULATION satisfies its specs trivially since all it does is to create an empty list and then call SEQ-ACC-1 with this list as one argument and the temporal collection as the other.

Now REASON assumes that all internal occurrences of SEQ-ACC-1 satisfy their specs. It follows that ACCUM-OP produces a list containing all members of the CURRENT-ACCUMULATION input plus the one additional element which is its other input. This element is a member of the temporal collection. The inputs to the recursive call of SEQ-ACC-1 are the new accumulation and the remaining members of the temporal collection. By the specs of SEQ-ACC-1 this will return a list containing all the objects which were members of its list input plus all the objects which were members of the temporal collection input. Thus, it produces a list of all the members of the temporal collection input to the main program. By induction REASON can conclude that SEQ-ACC-1 satisfies its specs.

It follows that a plan which is a functional composition of TREE-TRAVERSAL with SEQUENTIAL-ACCUMULATION will produce a list of the nodes of the tree. If a FILTERING plan is put between them so that only the terminal nodes of the tree are members of the temporal collection output of the filter, then we will have a plan for computing the fringe of the tree. It remains to show, however, that our original FRINGE program can, indeed, be looked at in this way.

Chapter 9: The Recognition Paradigm

The apprentice depends on a library of pre-analyzed plan fragments for use in analysis by inspection. So far, I have presented a formalism which allows these fragments to be stated in a general and abstract manner. Fragments like *TREE-TRAVERSAL* OF *SEQUENTIAL-ACCUMULATION* can be easily applied to any programming language and to a variety of data-structures and syntactic constructs which represent similar temporal behaviors.

The apprentice uses pre-analyzed plan fragments to help explain *parts* of programs, matching sections of program code to particular fragments. If all parts of a program are mapped onto some plan fragment and if these fragments are connected in coherent ways and if the entire artifact so constructed implements a desired behavior, we can then say that the program has been analyzed. In such cases the plan fragments have been used as if they were proof rules (of a rather macro character), showing that their preconditions hold and asserting their conclusions.

However, it will often be the case that only some of the code can be mapped onto fragments in the library of plans. In such cases it would be erroneous to assert that the program had been verified (or totally understood). Nevertheless, the program has been partially understood and half a loaf is better than none.

The recognition process proceeds as follows. First, a simple language dependent process translates the source language program text into a plan diagram. Such programs have been developed for LISP by Rich [Rich & Shrobe, 1976] and for FORTRAN by Waters [Waters, 1977]. This diagram, called the *surface plan*, is typically quite unstructured, having only that rudimentary segmentation which is implied by primitives such as IF-THEN-ELSE, DO, COND, PROCEDURE-CALL etc. Data-flow links are deduced by a symbolic interpretation (developed by Rich in [Rich & Shrobe, 1976]) of primitives such as assignment to variables, nested function applications etc. After this translation to plan diagrams, the raw code is not consulted, although links to it are maintained.

A *recognition mapping* of this surface plan consists of an aggregation of some of its original segments into larger segments, a mapping of these to the segments of a library plan such that all plan-type and spec-type constraints of the library plan are satisfied by the surface plan segments, and such that the data-flow, control-flow and conditional-control-flows of the surface plan are consistent with those of the library

plan. If such a recognition mapping can be constructed, it then follows that any property of the deep plan is also true of the corresponding surface plan.

Actually, the above conditions are slightly too strong. In constructing a recognition mapping it is allowable to ignore some of the inputs or outputs of surface plan segments. Data-flows connected to such inputs and outputs must also be ignored as must those spec clauses which mention such objects. Within the surface plan, sub-segments which are connected to ignored inputs must also be ignored. This allows us to separate a segment's behavior into those parts which involve a set of objects under consideration and those which do not.

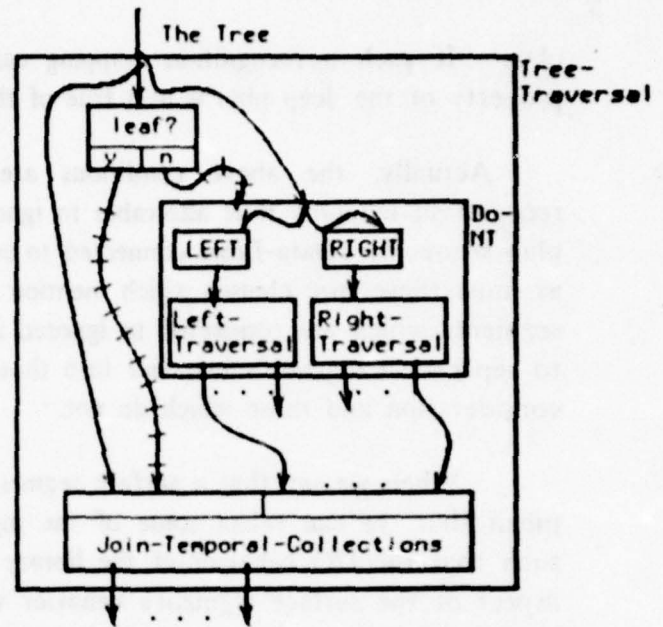
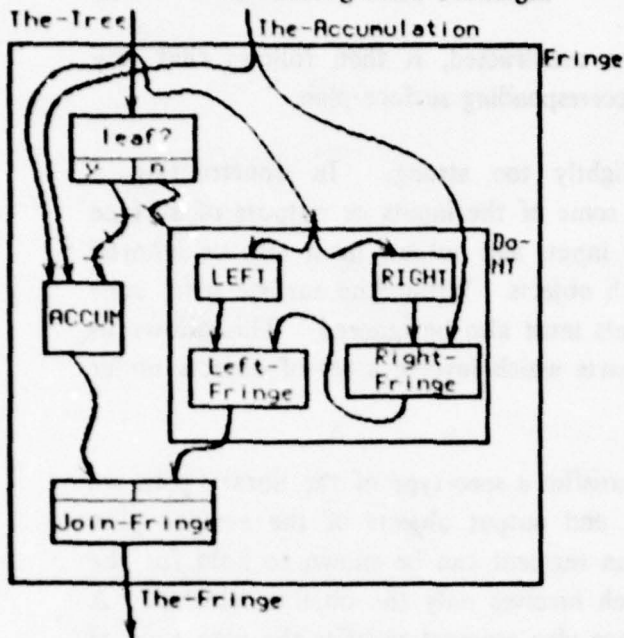
When we say that a surface segment satisfies a spec-type of the library plan we mean that we can select some of the input and output objects of the surface plan such that the I/O behavior of the library plan segment can be shown to hold for the aspect of the surface segment's behavior which involves only the objects selected. A similar principle applies to saying that a surface plan segment satisfies the plan type of its corresponding library plan segment.

Notice that a recognition mapping is not required to map every surface plan segment into a library plan segment. However, if a set of recognition mappings have been constructed such that each surface plan segment is in the domain of at least one completely constructed recognition mapping, then we say that the surface plan has been *completely recognized*.

I will now proceed to show how the original FRINGE program can be completely recognized as a composition of TREE-TRAVERSAL, LEAF-FILTERING, and SEQUENTIAL-ACCUMULATION. Following a sketch of this recognition process I will indicate how to extend the notions of recognition so as to gain greater abstraction power.

The construction of the recognition mapping depends on an inductive argument. We wish to show that by ignoring the ACCUMULATION input, we may regard FRINGE as an instance of the TREE-TRAVERSAL plan. Ignoring the ACCUMULATION input forces us to ignore the ACCUM-OP segment, as well as the ACCUMULATION input to the two recursive calls. We can then construct the straightforward recognition mapping of TEST-LEAF in FRINGE to TEST-LEAF in TREE-TRAVERSAL, LEFT to LEFT, RIGHT to RIGHT, and (RIGHT) LEFT-FRINGE to (RIGHT) LEFT-TREE-TRAVERSAL. Each of these segments except the two FRINGE segments satisfy their spec type constraints trivially. For the two FRINGE segments the induction hypothesis is invoked giving the desired result.

For Complex Program Understanding

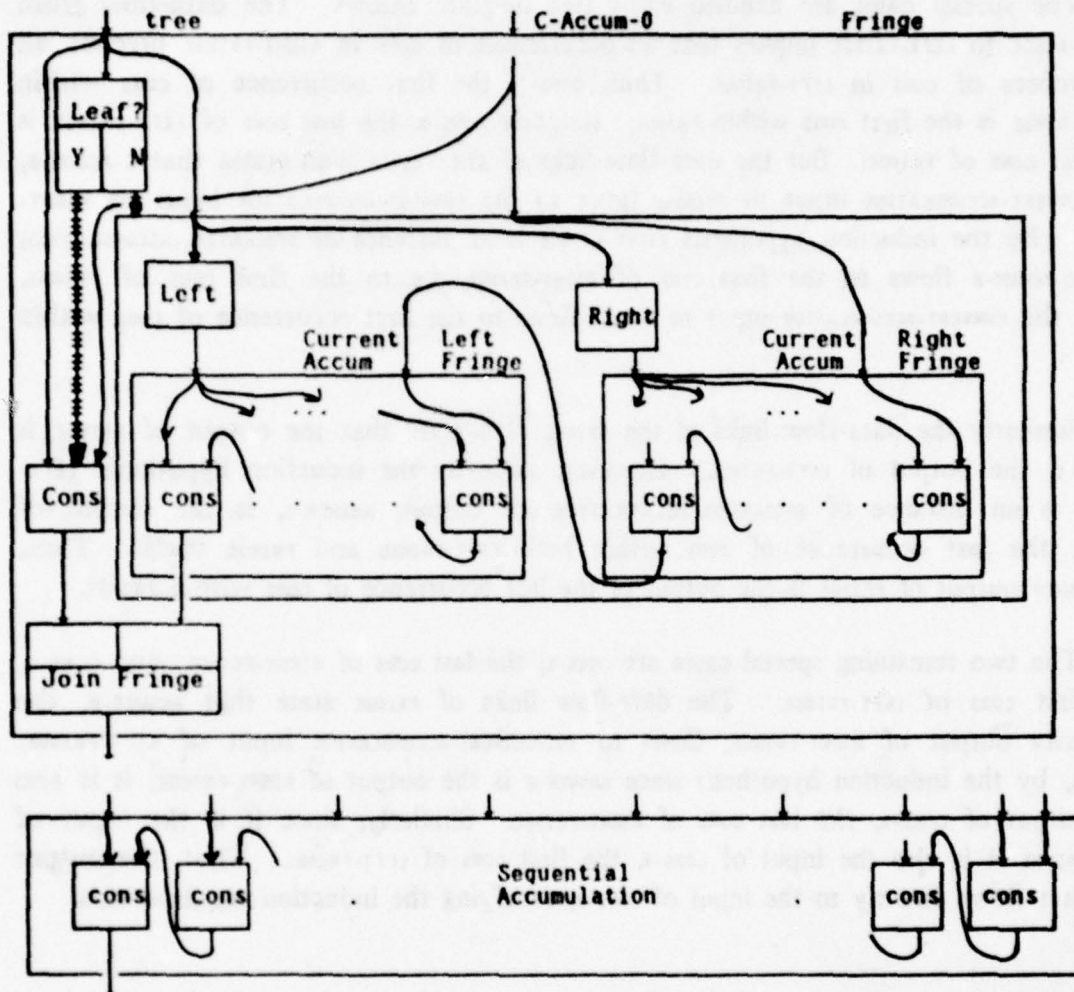


Recognition of Fringe as a Tree Traversal

This partial recognition tells us that the temporal collection generated by the occurrence set of type `LEAF?` within `FRINGE` includes exactly the nodes of the input tree. We now have to construct a partial recognition for the `SEQUENTIAL-ACCUMULATION` plan.

The recognition mapping identifies the occurrences of the `CONS` segment in `FRINGE` with the `ACCUM-OPS` of `SEQUENTIAL-ACCUMULATION`; the data-flows from `TEST-LEAF` to the `CONS` are mapped onto the temporal collection data-flow to the `ACCUM-OPS`. The remaining data-flows of the library plan for `SEQUENTIAL-ACCUMULATION` then require us to show that the first `CONS` in `FRINGE` receives an empty list as its input, that the output of each occurrence of `CONS` flows to the next occurrence, and that the output of the last occurrence of `CONS` flows to the output of the whole segment.

The proof of these claims is a straightforward (plan-type) induction. We assume that the occurrence set of type `CONS` within each internal application of type `FRINGE` is in fact a `SEQUENTIAL-ACCUMULATION`. There are two cases to consider. If the tree input to `FRINGE` is a terminal, then there is exactly one occurrence of `CONS`; its inputs are the terminal node itself and the `CURRENT-ACCUMULATION` input of `FRINGE`; its output is the output of `FRINGE`. Thus, in this case all the requirements are met and we may regard the occurrence set of type `CONS` as a `SEQUENTIAL-ACCUMULATION`.



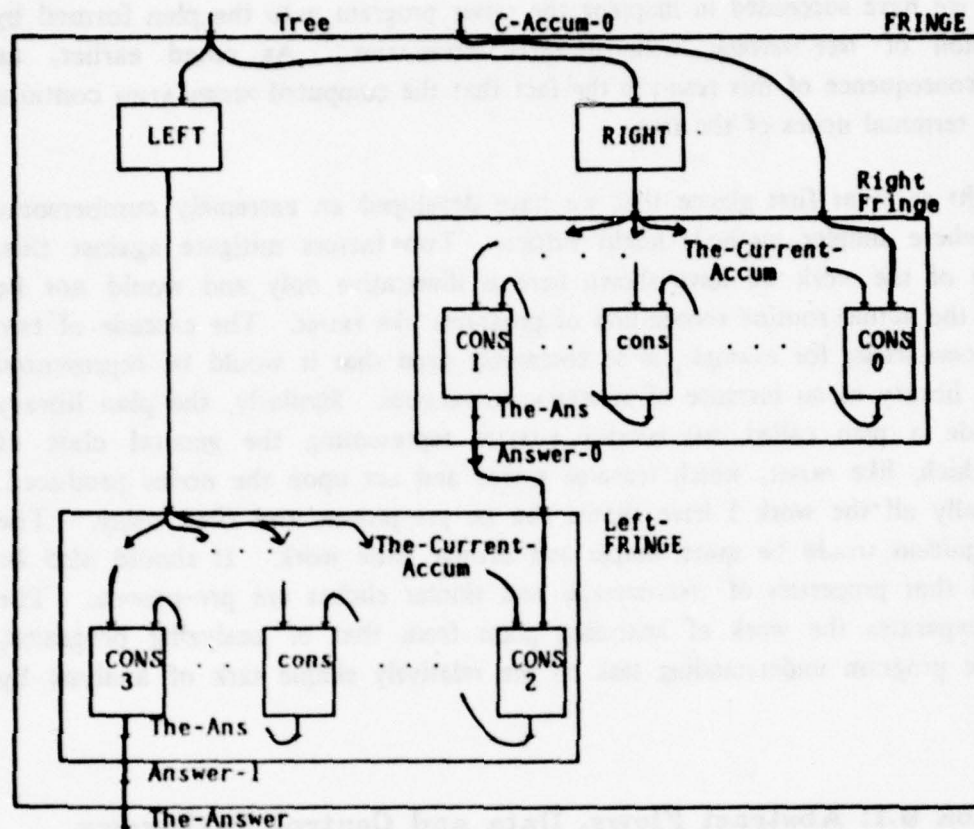
Recognition of Fringe as Sequential Accumulation

In the second case, the tree input to FRINGE is non-terminal and there are two recursive occurrences of type FRINGE. We can assume the induction hypothesis for both the LEFT-FRINGE and RIGHT-FRINGE occurrences of FRINGE. We now construct a case analysis. By the induction hypothesis the occurrence set of type cons within each internal application of type FRINGE is a SEQUENTIAL-ACCUMULATION. This leaves four special occurrences of cons, namely the first and last occurrences within LEFT-FRINGE and RIGHT-FRINGE. All other occurrences of cons cascade their outputs to the next occurrence and receive one of their inputs from the previous occurrence.

The special cases are handled easily (see diagram below). The data-flow from RIGHT-FRIDGE to LEFT-FRIDGE implies that all occurrences of CONS in RIGHT-FRIDGE precede all occurrences of CONS in LEFT-FRIDGE. Thus, CONS-0, the first occurrence of CONS within RIGHT-FRIDGE is the first CONS within FRIDGE; similarly CONS-3, the last CONS of LEFT-FRIDGE is the last CONS of FRIDGE. But the data-flow links of the FRIDGE plan states that C-ACCUM-0, the CURRENT-ACCUMULATION input to FRIDGE, flows to the CURRENT-ACCUMULATION input to RIGHT-FRIDGE. By the induction hypothesis RIGHT-FRIDGE is an instance of SEQUENTIAL-ACCUMULATION, thus C-ACCUM-0 flows to the first CONS of RIGHT-FRIDGE, i.e. to the first CONS of FRIDGE. Thus, the CURRENT-ACCUMULATION input to FRIDGE flows to the first occurrence of CONS within FRIDGE.

Similarly the data-flow links of the FRIDGE plan state that the output of FRIDGE is ANSWER-1, the output of LEFT-FRIDGE. However, since by the induction hypothesis LEFT-FRIDGE is an instance of SEQUENTIAL-ACCUMULATION, its output, ANSWER-1, is the output of CONS-3, the last occurrence of CONS within both LEFT-FRIDGE and FRIDGE itself. Thus, THE-ANSWER output of FRIDGE is the output of the last occurrence of CONS within FRIDGE.

The two remaining special cases are CONS-1, the last CONS of RIGHT-FRIDGE, and CONS-2, the first CONS of LEFT-FRIDGE. The data-flow links of FRIDGE state that ANSWER-0, the THE-ANSWER output of RIGHT-FRIDGE, flows to THE-CURRENT-ACCUMULATION input of LEFT-FRIDGE. Again, by the induction hypothesis since ANSWER-0 is the output of RIGHT-FRIDGE, it is also the output of CONS-1, the last CONS of RIGHT-FRIDGE. Similarly, since it is the input of LEFT-FRIDGE it is also the input of CONS-2, the first CONS of LEFT-FRIDGE. Thus, the output of CONS-1 flows directly to the input of CONS-2, satisfying the induction requirements.



Details of Recognition: Fringe as Sequential Accumulation

To complete the recognition of FRINGE as a TREE-TRAVERSAL composed with a SEQUENTIAL-ACCUMULATION, the final thing we need to show is that the temporal collection output of TREE-TRAVERSAL flows to the temporal collection input of SEQUENTIAL-ACCUMULATION. In terms of the recognition mapping which has been constructed, this means that the temporal collection generated by the occurrence set of type LEAF? should flow to the occurrence set of type CONS. Again this is a direct result of a simple inductive argument. If the input tree is a terminal then the result is trivial, there is exactly one data-flow between TEST-LEAF, the one occurrence of type LEAF? and the one occurrence of type CONS. In the non-terminal case, the induction hypothesis says that there is a temporal collection data-flow between the occurrence set of type LEAF? and the occurrence set of type CONS in both LEFT-FRIDGE and RIGHT-FRIDGE. But if the input tree is non-terminal, these are the only occurrences of type CONS within FRINGE. It follows that the temporal collection input to the occurrence set of type CONS within FRINGE is exactly the temporal collection generated by the occurrence set of type LEAF? within FRINGE.

For Complex Program Understanding

Thus, we have succeeded in mapping the `FRINGE` program onto the plan formed by a composition of `TREE-TRAVERSAL` with `SEQUENTIAL-ACCUMULATION`. As noted earlier, an immediate consequence of this result is the fact that the computed `ACCUMULATION` contains exactly the terminal nodes of the tree.

It might seem at first glance that we have developed an extremely cumbersome technique where simpler methods might suffice. Two factors mitigate against this. First, much of the work we have shown here is illustrative only and would not be required in the actual routine recognition of programs like `FRINGE`. The cascade of two `SEQUENTIAL-ACCUMULATIONS`, for example, is so commonly used that it would be represented in the plan library as an instance of `SEQUENTIAL-ACCUMULATION`. Similarly, the plan library could include a plan called `TREE-TRAVERSAL-&-ACTION` representing the general class of programs which, like `FRINGE`, which traverse a tree and act upon the nodes produced. Thus, virtually all the work I have shown can be pre-proven and filed away. The actual recognition would be quite simple and involve little work. It should also be remembered that properties of `TREE-TRAVERSAL` and similar cliches are pre-proven. The apprentice separates the work of analyzing plans from that of analyzing programs, reducing the program understanding task to the relatively simple task of analysis by inspection.

Section 9.1: Abstract Flows, Data and Control Pathways

The strength of this method depends on the ability to abstract out details of the code in order to view the program as an instance of a more abstract but better understood artifact. Abstraction techniques allow the plans to achieve greater generality. So far we have seen techniques for abstracting procedural behavior (specs), various issues of data-flow (temporal collections) and control-flow (temporal control sequences). I will now add a further abstraction to the repertoire used in the plan diagram notation. This new feature called control-and data-pathways will allow the apprentice to recognize programs as instances of plans to which they bear little immediate resemblance. This will allow us to represent commonalities at a more abstract level.

Consider the following program which computes the fringe of a tree using a breadth first traversal.

```
(defun Qfringe(tree)
  (prog (Acc Node Q)
    (setq Q (Empty-Queue)) (Enqueue tree Q))
  1p (cond ((Queue-Empty? Q)(return Acc))
    (t (setq Node (Dequeue Q))
      (cond ((Leaf? Node)(setq Acc (cons Node Acc)))
        (t (Enqueue (left Node))
          (Enqueue (right Node))
            (go 1p))))))
```

A close examination of the temporal behavior of this program reveals that it is quite similar to the FRINGE program of the previous section. In both programs there is a traversal of the nodes of the tree, a filtering of these nodes to select the leaf nodes, and an accumulation of the leaves. This accumulation is the output of both processes. In fact, the only significant difference between the two programs is the order of traversal of the nodes of the tree. However, since the library plan for TREE-TRAVERSAL makes no commitment to the order of traversal, it ought to be possible to recognize QFRINGE as an instance of the "fringe plan".

On the other hand, there are obvious superficial differences. FRINGE is doubly recursive in its surface syntax, whereas QFRINGE is a loop (singly recursive). QFRINGE has an explicit queue while FRINGE has no similar explicit data-structure. Given the superficial clues present in QFRINGE, any reasonable recognition process would first guess that QFRINGE is an instance of the QUEUE-AND-PROCESS plan shown below.

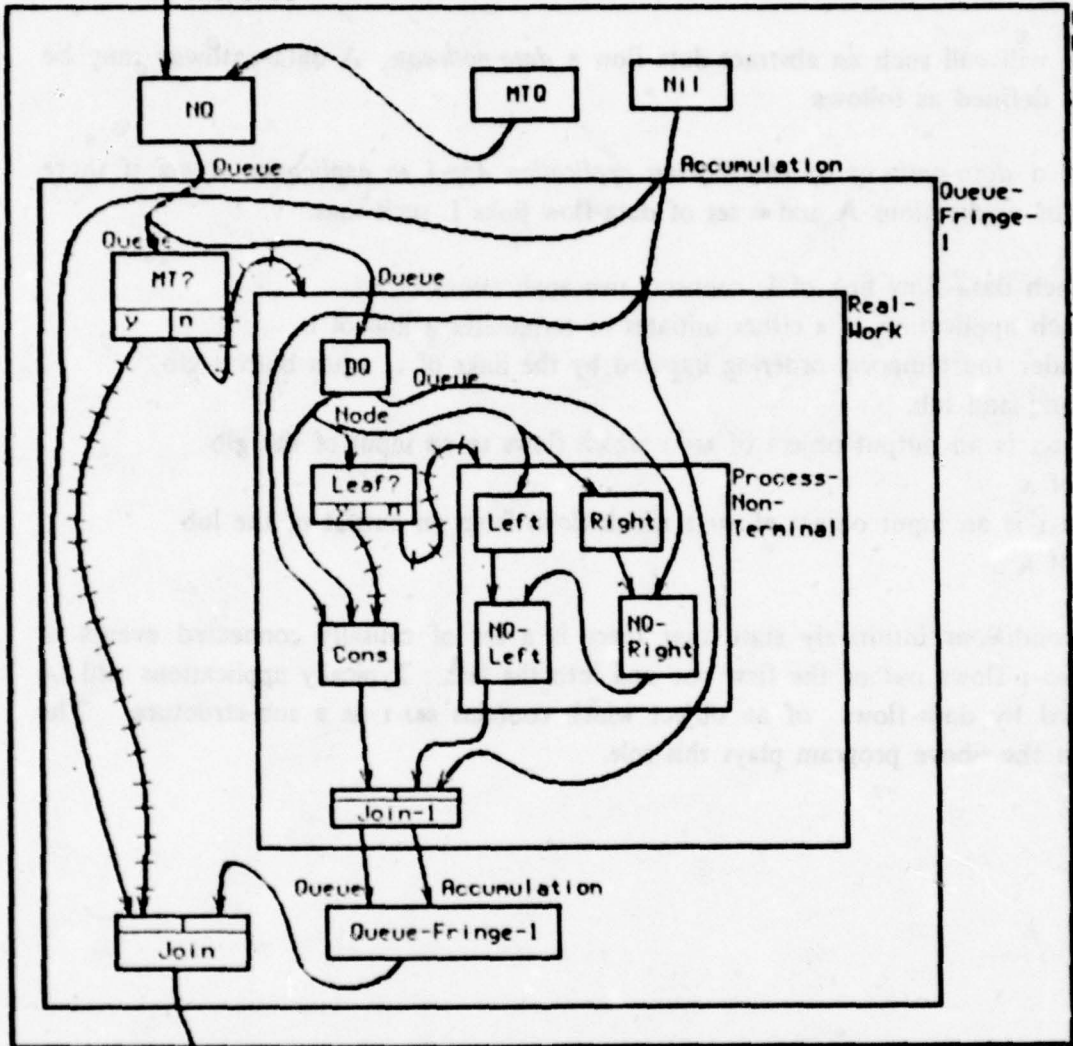
transmission is of little concern as long as we can believe that no significant property of the transmitted object will be lost in the process.

We will call such an abstract data-flow a *data-pathway*. A data-pathway may be formally defined as follows:

There is a *data-pathway of Obj-1 from application App-1 to application App-2* if there is a set of applications A and a set of data-flow links L such that:

- (i) Each data-flow link of L connects two applications of A .
- (ii) Each application of A either initiates or terminates a link of L .
- (iii) Under the temporal ordering imposed by the links of L , A has both a glb and a lub.
- (iv) $obj-1$ is an output object of $app-1$ which flows to an input of the glb of A .
- (v) $obj-1$ is an input object of $app-2$ which flows from an output of the lub of A .

These conditions intuitively state that there is a set of causally connected events in which $obj-1$ flows out of the first one and into the last. Typically applications will be connected by data-flows of an object which contains $obj-1$ as a sub-structure. The queue in the above program plays this role.



Breadth First Fringe Program Using a Queue and Process Plan

Breadth First Fringe Program Using A Queue and Process Plan

It is easy to prove by computational induction that for any object which is a member of the input queue of an occurrence of `QUEUE-AND-PROCESS-1` there will be a data-pathway to an occurrence of `PROCESS-MEMBER`. It is also easy to show that `QFRINGE` is an instance of `QUEUE-AND-PROCESS`, mapping `QFRINGE-1` onto `Q-AND-PROCESS-1` and `PROCESS-NODE` onto `PROCESS-MEMBER`. Thus, for each member of the queue in an application of `QFRINGE-1` there will be a data-pathway via a sequence of enqueues and dequeues to an occurrence of `PROCESS-NODE`.

If we regard these data-pathways as data-flows, we may then construct a recognition mapping between `QFRINGE` and `FRINGE`. If `TREE-0`, the input node, is non-terminal then the `TEST-LEAF` segment in `QFRINGE` will bring control to `PROCESS-NON-TERMINAL`. `LEFT` will extract `LEFT-0`, the left node of `TREE-0`, and `NOL` will make `LEFT-0` a member of the queue. `RIGHT` and `NOR` will act similarly on `RIGHT-0`, the right node of `TREE-0`. Control now passes to `QFRINGE-1` (i.e. we return to the beginning of the loop), with both `LEFT-0` and `RIGHT-0` members of the queue. By our remarks above, each of these flows via a data-pathway to another occurrence of `PROCESS-NODE`. Let us call these occurrences `PN-1` and `PN-2` respectively. We can map `PN-1` onto `LEFT-FRINGE` and `PN-2` onto `RIGHT-FRINGE`, thus completing the recognition of `QFRINGE` as a `FRINGE` program.

The use of pathways becomes quite important in more complicated queue and stack based programs such as procedural deduction systems which rely on pattern directed invocation. In these programs, processes communicate by making assertions in a data-base possibly triggering other programs into execution. The notion of a pathway makes the description of this mechanism far more concise than would be possible otherwise. Furthermore, it allows the system to understand such demon-based programs in terms of the communications between processes rather than in terms of the mechanism of communication. Pathways are analogous to the *sometimes* notion of Manna and Waldinger [Manna & Waldinger, 1976] in that they speak of control reaching a certain point at some future time, rather than immediately; however, I use pathways as an abstraction tool which transforms one plan diagram into a second, more easily recognized diagram.

Because pathways allow this greater flexibility, all flow statements of a library plan may be matched by pathways implemented in the surface plan. Library plans are stated in terms of data abstractions, specs, purpose links, data and control pathways.

Section 9.2: Summary

The methodology I have outlined relies on developing a library of pre-proven plan fragments which capture substantial parts of the knowledge of an expert programmer. Once such fragments have been catalogued, the effort of program analysis may be reduced to that of recognition. I have not discussed what heuristics would guide such a recognition system, interesting work in that direction is being conducted by Rich [Rich, 1977] and Waters [Waters, 1977]. Instead I have concentrated on the how such a process would interact with the reasoning capabilities of the program analysis system.

Our method might be challenged on the grounds that it involves as much work as more standard approaches to program verification. However, this work is factored in two ways which are highly significant. First, we divide the task into (1) Pre-proving frequently used *standard plan fragments* whose logical analysis need never be repeated, and (2) Recognizing the occurrences of these fragments within more complex programs.

Second, the recognition process itself is factored into many discrete steps, such as: (1) Showing that each of the proposed segments of the surface plan satisfies the type constraints of the library plan, (2) Showing that the data-flows of the surface plan implement the data-pathways of the conceptual plan. While the total amount of work involved might be substantial it is separated into "bite sized" pieces; furthermore, the framework allows the reasoning system to self-consciously concentrate on the particular goal at hand at that moment. While attempting one particular recognition all other parts of the program can be ignored.

Chapter 10: Description of Data-Structures

So far I have presented methods of describing various components of programming knowledge; I have also shown the various reasoning techniques used in **REASON** to operate on these descriptions. In this chapter I will develop a language for describing the static properties of data-structures. In the next chapter (which will discuss side-effects) I will describe **REASON**'s methods for reasoning about how properties of data-structures change.

Knowledge about data-structures is a key component of the knowledge base needed by the programmer's apprentice system. This knowledge is used by the synthesis and recognition systems in a declarative form. In the reasoning part of the system, descriptions of data-structures are used in a more active or procedural manner.

Data-structures are perhaps the most flexible items in the apprentice's knowledge base; programmers routinely devise new data-structures and new methods of implementing them. It is crucial that a convenient language be developed for the description of data-structures. This language allows the programmer to tell the apprentice about new data-structures, their decomposition into parts and the constraints which these parts must satisfy. Finally, it allows the programmer to devise new relationships which might be true of the new data-structure and to provide the apprentice with definitions of these new relations.

The data-description language has two main features. First, it is syntactically declarative, allowing the programmer to describe objects without having to know the rule-based syntax of the deductive system. The declaratives are translated by **REASON** into rules which actually make the deductions. Second, the description language allows hierarchical knowledge structures and inheritance of properties. An **ALIST** can be described as a special kind of **LIST** which in turn is described as a special kind of **RECURSIVE-STRUCTURE**. Properties are described at the most general level possible. Typically the programmer need only state that the structure he is designing is a special case of some other (or of several other) structure(s). Each aspect of the inherited behavior of the newly defined structure is then mapped down from the parent structure(s). Usually, only a few new properties are involved in the definition of any new object.

Section 10.1: The Data-Description Language

At the lowest level, a data-structure is something which has specific *parts* subject to certain constraints. For example, a list is usually described as having a *FIRST* and a *REST* subject to the restriction that the rest must be either another *LIST* or a terminator such as *NIL*. A *BINARY-TREE*, similarly, has a *LEFT* and a *RIGHT* where both are required to be either *BINARY-TREES* or *TERMINALS*. These ideas are represented by two types of assertions: *part* and *type-restriction*.

```
(Part <object-type> <part-name>)
(Type-restriction (<part-name> <object-type>) <object-type>)
```

For example,

```
(Part List First)
(Part List Rest)
(Type-restriction (Rest List) List)
```

Object-types are sets of similarly structured objects such as *LIST* or *BINARY-TREE* and are subject to the normal set operations of union, intersection, set difference, etc. In particular, we use the subset relation extensively to build a hierarchy of knowledge. For example, *LISTS* are *LINEAR-OBJECTS*, and *PROPERTY-LISTS* are *LISTS*. This is stated as follows:

```
(Subset list linear-object)
(Subset property-list list)
```

Two useful pieces of information are extracted from such statements by *REASON*'s rules. First, anything which is a subset of an object-type is itself an object-type; second, anything which is a member of a subset of an object-type is a member of the larger type as well.

```

(rule
  ((:a (subset :subtype :supertype)))
  (assert '(object-type :subtype)
    '(subset-implies-type :a)))

(rule ((:a (subset :subtype :supertype))
      (:b (type :object :subtype)))
  (assert '(type :object :supertype)
    '(type-chain :a :b)))

```

Object-types are often combined to yield new types. For example, we might want to have an object-type which includes both `EMPTY-` and `NON-EMPTY-LISTS`. This object-type might be called `LIST`. However, it is also useful to distinguish between `EMPTY-` and `NON-EMPTY-LIST`; therefore, we also have the two specialized object-types called `EMPTY-LIST` and `NON-EMPTY-LIST`. Obviously, `LIST` is the union of the other two object-types; in addition the intersection of `EMPTY-LIST` and `NON-EMPTY-LIST` is the null set. Such a situation is so common in defining object-types that I have distinguished it with the special name *partition*. We write this as follows:

```
(partition List into: (Empty-List Non-Empty-List) )
```

from which it follows that:

```

(Subset Empty-List List)
(Subset Non-Empty-List List)
(Union (Empty-List Non-Empty-List) List)
(Intersection (Empty-List Non-Empty-List) Null-Set)

```

It is usually insufficient to know only that an object-type is partitioned; we also need to know how to distinguish between the sub-types. The partition of `LIST` into `EMPTY` and `NON-EMPTY` exhibits a very frequent and common method of distinction, namely that one of the sub-types has a more detailed part structure. Sometimes, however, a more involved criterion is used to distinguish sub-types. For example, we may partition `LISTS` into `CYCLIC-LIST` and `ACYCLIC-LIST`

using a much more complicated criterion. Two syntactic extensions to the `partition` statement are provided to facilitate the stating of these restrictions.

The first of these is the *allows* clause, written as follows

```
(Partition list
  info: (empty-list non-empty-list)
  (allows: non-empty-list (first rest)))
```

The *allows* clause says that the named sub-type (NON-EMPTY-LIST, in this case) includes the part names mentioned in its set of part names; furthermore, this is a distinguishing characteristic within the given partition. In the example above this means that NON-EMPTY-LISTS can be distinguished from EMPTY-LISTS by the presence of either a FIRST part or a REST part.

We can now give a simple description of LIST:

```
(Partition list
  info: (empty-list non-empty-list)
  (allows: non-empty-list (first rest)))
(Part List First)
(Part List Rest)
(Type-Restriction (Rest List) List)
```

similarly we can give a description of BINARY-TREES as follows:

```
(Partition Binary-Tree
  info: (Terminal Non-Terminal)
  (allows: non-terminal (Left Right)))
(Part Binary-Tree Left)
(Part Binary-Tree Right)
(Type-Restriction (Left Binary-Tree) Binary-Tree)
(Type-Restriction (Right Binary-Tree) Binary-Tree)
```

If the criterion which distinguishes between sub-types is more complex, we express this with a *dividing criterion* clause. For example, we can tell ACYCLIC-LISTS from CYCLIC-LISTS by the absence of any sub-list which is a sub-list of itself. This is stated as follows:

```
(Partition list
  info: (acyclic-list cyclic-list)
  (dividing-criterion:
    (For-All (:sub) (sub-list acyclic-list :sub)
      {not (proper-sub-list :sub :sub)})))
```


The data-description language uses several notational conventions which are shown in this example. The dividing criterion clause above mentions `ACYCLIC-LIST`. This use of a *type name* within a statement of the data-description language is an implicit quantification over any object of that type. I will explain this notation further in the next section. In the case of the `DIVIDING-CRITERION` clause, however, the meaning is quite simple. Any object of the partitioned type (say `LIST`) which satisfies the `DIVIDING-CRITERION` (the `FOR-ALL` statement) can be deduced to be of the sub-type mentioned in the `CRITERION` (`ACYCLIC-LIST`).

Now let us move on to other definitional statements. Closely related to the notion of `PART` is that of `INDEXED-PART` which is used to describe objects like arrays having many similar sub-structures distinguished by numerical indexing rather than by name. The index is permitted to be any tuple of integers, but in practice I will almost always be talking about single dimensioned `INDEXED-STRUCTURES`. `INDEXED-PARTS`, like `PARTS`, are subject to type restrictions. This is written as follows:

```
(Indexed-Part (object-type) (indexed-part-name))
(Type-Restriction ((indexed-part-name) (object-type)) (object-type))
```

Thus an array of integers would be described as follows:

```
(Object-Type Integer-Array)
(Indexed-Part Integer-Array Item)
(Type-Restriction (Item Integer-Array) Integer)
```

Some restrictions, however, are of a type which is difficult or awkward to describe as type restrictions. For example, the object used to index an `INDEXED-STRUCTURE` is required not only to be an integer tuple of the appropriate "a-arity" (a type restriction), it is also required to be within the correct bounds. Describing this as a type restriction would require the creation of an object-type for every range of integers. Instead these more complex restrictions may be stated directly with a *require statement*:

```
(Require
  (Item Integer-Array index object)
  (Index Integer-Array index))
```

which states that if an object is used as the selecting position of an INDEXED-PART statement, then it must be a valid index of the object whose component it selects. The notion of INDEX must, of course, be defined elsewhere (see later section on relation definitions). *Require* statements are invariants which state that any time their first clause is true, their second clause must be as well. Thus, there are two ways to make use of the information encoded in such a statement. The first is to deduce the consequent from the antecedent; the second is to require that the consequent hold any time the antecedent is realized through a side-effect.

TYPE-RESTRICTIONS may be regarded as a special case of REQUIRE statements in which the antecedent is a PART OF INDEXED-PART statement and in which the consequent is an OBJECT-TYPE statement. In fact, both types of statements are translated into REASON rules in a rather straightforward manner which makes this clear. For example, the above REQUIRE statement leads to the following:

```
(Rule ((:a (Item :is :index :obj))
      (:b (type :is indexed-structure)))
      (Assert '(Index :is :index) '(require :a :b c)))
```

where c is the fact-name of the REQUIRE statement. Similarly a TYPE-RESTRICTION would be translated into the following:

```
(Rule ((:a (Left :bt :node))
      (:b (type :bt binary-tree)))
      (Assert '(Type :node binary-tree) '(type-rest :a :b c)))
```

where c is the fact-name of the TYPE-RESTRICTION statement.

Given these means for describing the component structure of an object-type we may now go on to define properties and relations of data-objects. Consider a BINARY-TREE; we would like to describe what it means to be a NODE of such a tree. We do this with a simple recursive definition: a NODE of a NON-TERMINAL BINARY-TREE is either the TREE itself or a NODE of the LEFT of the TREE or a NODE of the RIGHT of the tree. The only NODE of a TERMINAL BINARY-TREE is the TREE itself. this is represented as follows:

```

F-1 (Relation-Definition (node non-terminal binary-tree)
      (<=> (or (id non-terminal binary-tree)
                (node [left non-terminal] binary-tree)
                (node [right non-terminal] binary-tree))))

F-2 (Relation-Definition (node terminal binary-tree)
      (<=> (id terminal binary-tree)))

```

Notice that assertions of the data-description language do not use REASON variables (e.g. NON-TERMINAL) but only simple identifiers (e.g. NON-TERMINAL). This is a notational convenience which is possible since REASON translates these statements into REASON rules which do use variables.

There is a second notational shorthand involved here. Each IDENTIFIER (i.e. everything other than the logical operators and the predicate names) used in a data-description statement is an OBJECT-TYPE (e.g. BINARY-TREE, NON-TERMINAL, etc.). This allows relations to be *poly-morphic*, i.e. one relation (such as MEMBER) may apply to many different pairs of object-types. MEMBER, for example, is a relation which holds between LISTS and OBJECTS, HASH-TABLES and ENTRIES, ALISTS and DOTTED-PAIRS, etc. The use of object-types as the IDENTIFIERS of the RELATION-DEFINITION restricts the applicability of the definition to exactly those cases where the objects involved satisfy the type constraints implied by the IDENTIFIERS. (IDENTIFIERS may actually be subscripted object-types such as NON-TERMINAL-1 in those cases where there is more than one object of a type mentioned within a single relation). It is important to remember that this notational principle applies only to statements of the data-description language, not to REASON's rules. When data-description assertions are translated into the rules of the reasoning system, the requirements implicitly represented by the IDENTIFIERS of the data description statement are made explicit as triggers of the rule.

RELATION-DEFINITIONS are used in a number of ways in REASON, corresponding to the various ways in which implications can be used in logic systems. The simplest of these is the substitution of a right hand side for the left. REASON translates the above definition into the following rule which will make this substitution if requested to do so:

176 Description of Data-Structures

```
(Rule
  ((:e (expand ((node :nt :bt) :s))))
(Rule
  ((:a ((node :nt :bt) :s))
   (:b ((non-terminal :nt) :s))
   (:c ((binary-tree :bt) :s)))
  (Assert '(Or (:id :nt :bt)
                ((node s[left :nt] :s2 :bt) :s)
                ((node s[right :nt] :s2 :bt) :s))
    '(Rel-def :a :b :c F-1))))
```

In this rule there are two levels of invocation. The outer level is triggered by an explicit request to expand the assertion. This creates the inner rule which checks to see if, in fact, the assertion to be expanded is believed and then checks to see if the **TYPE-CONSTRAINTS** are satisfied. If so, the definition is asserted and justified by the triggering facts. However, if the **TYPE-CONSTRAINTS** are not satisfied then this rule represents an inappropriate definition and no assertion is made.

A second use of **RELATION-DEFINITIONS** is in the antecedent deduction of a relation such as **MEMBER** from the facts corresponding to the right hand side of the definition. For example, the following is the standard recursive definition for membership in a **LIST**.

```
F-3 (Relation-definition (Member List Object)
  (<=> (or (first list object) (member [rest list] object))))
```

This definition has the following antecedent use: if we know that an object is the **FIRST** of a **LIST**, or if we know that it is a **MEMBER** of the **REST** of the **LIST**, then we can infer that it is a **MEMBER** of the **LIST**. This corresponds to two rules which **REASON** creates from this definition:

```

(Rule
  ((:a ((first :l :o) :s))
   (:b ((type :l list) :s)))
  (assert '((member :l :o) :s) '(Rel-def :a :b F-3)))

(Rule
  ((:a ((rest :l :r) :s))
   (:b ((member :r :o) :s))
   (:c ((type :l list) :s)))
  (assert '((member :l :o) :s) '(Rel-def :a :b :c :d F-3)))

```

Notice that in the second rule REASON translated the reference expression [REST LIST] into a pattern in the trigger set of the rule. All reference expressions within the clauses of a RELATION-DEFINITION are handled in this way; recursively nested expressions are brought out to the top level. MEMBER in a HASH-TABLE is such a relation. A HASH-TABLE is an INDEXED-STRUCTURE in which each component (called a BUCKET) is a set, represented by some data-structure. There is a functional relationship called HASH which maps ENTRIES into INDICES of the TABLE. An ENTRY is a MEMBER of the TABLE if it is a MEMBER of the BUCKET into which it HASHES.

F-4 (Relation-Definition (member hashtable entry))
 (<=> (member [bucket hashtable [hash entry]] entry))

This definition is translated into the following rule.

```

(Rule
  (:a1 ((Member :b :e) :s))
  (:a2 ((type :e entry) :s))
  (:a3 ((type :b bucket) :s))
  (:a4 ((hash :e :index) :s))
  (:a5 ((bucket :t :index :b) :s))
  (:a6 (type :t hashtable) :s))
  (assert '((member :t :e) :s)
    '(rel-def :a1 :a2 :a3 :a4 :a5 :a6 F-4)))

```

Such rules are built by first replacing the reference expressions by equivalent existentially quantified statements and then transforming the statement in clausal form. Each disjunct of the resulting expression is then translated directly into a rule with the set of conjuncts forming the trigger set.

We should note that RELATIONS, like PARTS, are subject to TYPE-RESTRICTIONS. These restrictions are represented with the same notation as TYPE-RESTRICTIONS ON PARTS and INDEXED-PARTS. For example:

```
(Type-Restriction (node binary-tree) binary-tree)
```

which would be translated into a rule as shown earlier.

It is a rather common practice to build up new data-types by imposing restrictions on already existing types. A common example is that of a LISP association list (A-LIST for short) which is a LIST all of whose members are PAIRS. We indicate this as follows:

```
(defining-restriction
 (subset alist list)
 (type-restriction (member alist) pair))
```

Several distinct types of information are extracted from such a statement. First, there is the obvious SUBSET relation between ALIST and LIST, and the fact that the TYPE-RESTRICTION applies to the ALIST data-type. A simple rule adds these new facts:

```
(rule
 ((:a (defining-restriction :subset-fact :type-res-fact)))
 (assert :subset-fact '(def-rest :a))
 (assert :type-res-fact '(def-rest :a)))
```

However, there is a further piece of information in the DEFINING-RESTRICTION which is used in a consequent (backward chaining) manner. For example, if one wishes to show that an object is an ALIST, one possible strategy is to first show that it is a LIST and then show that it satisfies the defining restriction of having only PAIRS for MEMBERS. The following rule embodies this strategy.


```

(rule ((:a1 (goal ((Object-type :object alist) :s) for :goal in :context))
      (:a2 (defining-restriction (subset alist list) (type-restriction (member alist) pair))))
(propose-method
 '(Method :a1 (defining-restriction :a2)) '(def-rest :a1 :a2)
 (Assert
  '(conjunctive-goals (object-type :object list)
    ((for-all (:e1) ((member :object :e1) :s)
      ((object-type :e1 pair) :s)))
    (:a2)
    (((object-type :object alist) :s) . :goal)
    :context)
  '(defining-restriction :a1 :a2))))

```

where the CONJUNCTIVE-GOAL mechanism is explained in Chapter 4. Notice that if the system does conclude that an object is an ALIST, this conclusion will not depend on the control assertions created during the process but only on the OBJECT-TYPE assertion, the FOR-ALL assertion, and the DEFINING-RESTRICTION statement.

Section 10.2: Parameterized Object Descriptions

The statements I have shown so far allow us to state specific features of a data-structure; however, a descriptive method which "chunks" such statements into larger parcels of knowledge would be desirable. Such chunks of knowledge can then be organized into a library of programming skills [Rich & Shrobe, 1976], [Barstow, 1977] in such a way that generalities are conveniently captured and specialized as needed. The basic unit of description in this system will be a *parameterized object description*, which is a collection of statements describing the structure of a data-object. The parameters in an object description allow it to describe a family of related data-types. The object-types SET-OF-NUMBERS, SET-OF-LISTS, and UNRESTRICTED-SETS are all defined by the same parameterized description, only the choice of parameters is changed. This is quite similar to features found in CLU [Liskov, et al 1977] and ALPHARD [Wulf, et al 1976].

```

(Object-type-definition Set
  (parameters: Members-type (type-restriction: object-type))
  (partition Set into: (non-empty-set empty-set)
    (Allows: Non-Empty-set (Member)))
  (Relation (Member set members-type))
  (type-restriction (Member set) Members-type)
  (Relation (Union Set Set Set)
    (definition: (Union S-1 S-2 S-3)
      (<=> (Equiv (:e1) (or (member s-1 :e1)(member s-2 :e1))
        (member s-3 :e1))))))
)

```

The `equiv` quantifier above is merely an abbreviation for two universal quantifiers, one in each direction. `REASON`, in fact, treats these as abbreviations, translating them into the two quantifiers when the object-type definition is read in.

In this definition, `set` takes a single parameter which is required to be an object-type name. This name is then used at various points in the definition, for example in the `TYPE-RESTRICTION` statement. Such parameters act like the arguments to a macro generator; the type definition for `set` may be invoked with a specific parameter causing a more specific type to be created in which the parameter is instantiated. The following illustrates the syntax for such invocation:

```
(set (whose: (members-type: pos-number)))
```

which means the more specific object-type which consists of `SETS` whose `MEMBERS-TYPE` parameter is `POS-NUMBER`. Presumably, `POS-NUMBER` is a valid object-type and therefore this new type is as well. Inspection of the description indicates that this new object-type allows only `POS-NUMBERS` as `MEMBERS`, since the `MEMBERS-TYPE` parameter is used in the `TYPE-RESTRICTION` statement. Similarly,

```
(set (whose: (members-type: neg-number)))
```

would be the object-type of `SETS` of negative numbers. The convention is made that unspecified parameters assume a default value of *unrestricted*, so that the object-type specified as `set` with no parameters is treated as a `set` whose `MEMBERS` may be of any object-type. Notice that parameters are given with `TYPE-RESTRICTIONS`; in practice parameters are usually restricted to be either object-type names or numbers.

To avoid writing out the rather cumbersome type expressions above I have included an *is-a* statement to give a name to such expressions. Thus, we could write:

```
(object-type-definition Set-of-Pos-Numbers
  (is-a (set (whose: (members-type: pos-numbers))))))
```

Whenever an OBJECT-TYPE expression is used (either in IS-A clauses or elsewhere) an instantiation process is invoked to add this new object-type into the hierarchy of object-types. This invocation process consists of the following steps:

0. When the definition is first entered an object-type is created whose name is the name given in the definition statement and whose parameters are the default values. This will be referred to as the base-type, e.g. SET.
1. The new type is given a name. If used within an IS-A clause the name is taken from the clause. Otherwise, a default name is created.
2. The new type is declared to be a subtype of the base-type. DEFINING-RESTRICTIONS are added to reflect the effect of the specified parameters. For example, in the SET-OF-POS-NUMBERS object-type, the DEFINING-RESTRICTION is the TYPE-RESTRICTION OF MEMBERS TO POS-NUMBERS.
3. Parameter values are substituted for parameter names, and the new object-type name is substituted for the base-type name. Each statement within the definition is then processed. Those whose parameters are specified are asserted; those with default values are not.
4. If there were unspecified parameters, the partially instantiated object-type definition is added to the catalogue of object-type names. It, in turn, may serve as a prototype for further instantiation.

It is often convenient to be able to describe a related family of objects, using techniques similar to those above. For example, we might want a parameterized way of describing NUMERIC-INTERVALS, using the upper and lower bounds as parameters. Each instantiation of this description is a particular object (not an object-type, as above), but each such object differs only in minor regards from other such objects. Such descriptions are made with a *parameterized object description*.


```

(Object-definition Numeric-Interval
 (Parameters: (Upper-bound (type-restriction: Integer)
                          Lower-bound (type-restriction: Integer))
 (restriction: (Less-than Lower-bound Upper-bound)))
 (Is-a: (Set (whose: (Members-type: Integers))))
 (definition: (Numeric-Interval (whose: (upper-bound: U)(lower-bound: L)))
  = (:el) (And (less-than :el U)(greater-than :el L)))))

```

Like object-type descriptions, object descriptions take parameters subject to certain restrictions. The *is-a* clause allows us to give an object-type name to any object described by this parameterized description. Finally, the definition clause states an equality which defines what objects are named by the current description.

Object descriptions are processed as follows:

- (0) A new object-type is created using the name in the description as the *type-name*. (for example, *NUMERIC-INTERVAL* above).
- (1) If there is an *is-a* clause, the object-type name is asserted to a sub-type of the type named in the clause.
- (2) Any invocation of the description which leaves some parameters unspecified creates a new object-type (for example, *numeric-intervals* whose lower bound is 0).
- (3) An invocation which specifies all parameters creates an object which is asserted to be equal to the object specified in the definition clause. Finally, if the parameter-set for this object is a more specific set than the one used to create the next less specific object-type, then the object is asserted to be of that object-type. (For example, the numeric interval whose upper bound is 5 and whose lower bound is 0 would be asserted to be of the object-type of numeric intervals whose lower bound is 0 and whose upper bound is unspecified).

Section 10.3: Implementation and Virtual Objects

One object can be used to simulate the behavior of another. In fact, implementing more abstract data-structures using simpler ones is a great part of the effort in symbolic programming. Most often we use data-structures to represent basic mathematical concepts such as SETS, SEQUENCES, MAPPINGS, GRAPHS, and MULTI-SETS. However, there are a host of ways to build any of these representations. SETS may be represented by LISTS, ARRAYS, or HASH-TABLES; MAPPINGS by LISTS-OF-PAIRS, PAIRS-OF-ARRAYS, HASH-TABLES-OF-PAIRS, etc.

AD-A078 055

MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 6/4
DEPENDENCY DIRECTED REASONING FOR COMPLEX PROGRAM UNDERSTANDING--ETC(U)
APR 79 H E SHROBE
AI-TR-503

N00014-75-C-0643

NL

UNCLASSIFIED

3 OF 4

AD-A078055



The data-description language, therefore, requires two further extensions. First we must have a means of describing an implementation method; second, we need a way of stating that a particular abstract object is implemented using a particular method. Let us consider the classic example of implementing a STACK using an ARRAY and a POINTER. (What thesis could be complete without a STACK implemented as an ARRAY and a POINTER?) Let us say that we define a STACK to be an object with two parts, a TOP and a HISTORY, where the HISTORY is required to be another STACK (EMPTY or NON-EMPTY). We may describe this as follows:

```
(Implementation-Method Stack-As-Array
 (Abstract-Object: Stack)
 (Concrete-Objects:
   (Implementing-Array
    (type-restriction:
      (Array (whose: (members-type: [Members-type Stack])))))
   (Implementing-Pointer
    (type-restriction: Integer))
    (restriction: (Index Implementing-Array Implementing-Pointer)))
 (Represent: (Top Stack Object)
 (definition: (Top S 0)
   (*> (Item Implementing-Array Implementing-Pointer 0)))
 ...)
```

Before giving the rest of the method let us look at what we have so far. We say that one object may be represented by the behavior of a set of other objects using the ABSTRACT-OBJECT and CONCRETE-OBJECT clauses. These, of course, include TYPE and other restrictions which establish the pre-requisite conditions of the representation. We then describe how each of the undefined relations of the abstract object is mapped onto relations involving the concrete objects. This is done in the REPRESENT clause using equivalence statements like those used to define relations.

An implementation method implicitly defines a new object-type, namely objects of the abstract type which are implemented according to the specific method. This object-type is given the name of the IMPLEMENTATION-METHOD. Thus, we can build the following REASON rules which relate properties of objects of the abstract type to properties of objects of the concrete types:


```

(rule ((:f1 ((Object-type :S1 Stack-As-Array) :sit))
      (:f2 ((Implementing-Array :S1 :A) :sit))
      (:f3 ((Implementing-Pointer :S1 :I) :sit))
      (:f4 ((item :A :I :O) :sit)))
(assert '(((Top :S1 :O) :sit)
          '(Implementation :f1 :f2 :f3 :f4 :f5)))

(rule ((:f0 (expand ((top :S1 :O) :sit))))
      (rule ((:f1 ((Object-type :S1 Stack-As-Array) :sit))
              (:f2 ((Top :S1 :O) :sit)))
      (assert '(((item [Implementing-Array :S1]
                        [Implementing-Pointer :S1] :O) :sit)
                '(Expand-imp :f1 :f2))))

```

Notice that each concrete object in an IMPLEMENTATION-METHOD is given a name; this allows us to refer to the concrete objects by their description e.g. "the IMPLEMENTING-ARRAY of STACK-41". This bears such a similarity to the ways in which PARTS are used that I refer to this as an *implementation part*. When a new METHOD is described, assertions are added to the knowledge base stating what the implementation parts are

```

(Implementation-Part Stack-As-Array Implementing-Pointer)
(Implementation-Part Stack-As-Array Implementing-Array)

```

The corresponding TYPE-RESTRICTION statements are also added to the knowledge base. These are then used in exactly the same way as are PART statements and their TYPE-RESTRICTIONS. Also notice that the object-type implicitly defined by an IMPLEMENTATION-METHOD is a subtype of the abstract type. When a new method is entered into the system an assertion like the following is added to the data-base:

```

(Method-for Stack Stack-As-Array)

```

this triggers the following rule:

```

(Rule ((:f (Method-for :type-1 :type-2)))
      (assert '(:subset :type-2 :type-1)
              '(method-implies-subset :f)))

```

We still need to define how the HISTORY (SUB-STACK) part of the STACK is implemented. Intuitively we want to say that the HISTORY is the STACK implemented by (1) The SAME array and (2) The number which is one smaller than that of the current STACK. In the data-description language, therefore, we let the IMPLEMENTATION-METHOD name be a special operator. A predicate beginning with an IMPLEMENTATION-METHOD name

includes a an object name and a *whose* clause for each IMPLEMENTATION-PART-NAME. For example,

```
(Stack-As-Array (whose: (Implementing-Array: A1)(Implementing-Pointer: N1)) Stack-1)
```

means that stack-1 is the STACK implemented by the array a1 and the number n1, using the method STACK-AS-ARRAY. Thus, we may now describe how the history is represented by stating:

```
(Represent: (History Stack Stack)
 (definition: (History S1 S2)
  (<=> (Stack-As-Array (whose:
    (Implementing-Array: [Implementing-Array S1])
    (Implementing-Pointer:
      [plus 1 [Implementing-Pointer S1]])) S2)))
```

Using the techniques I have shown so far, this is turned into the following rule:

```
(Rule ((:f1 ((Object-type :S1 Stack-As-Array) :sit))
 (:f2 ((Implementing-Array :S1 :A) :sit))
 (:f3 ((Implementing-Pointer :S1 :N1) :sit))
 (:f4 ((Plus 1 :N1 :N2) :sit))
 (:f5 ((Object-type :S2 Stack-As-Array) :sit))
 (:f6 ((Implementing-Array :S2 :A) :sit))
 (:f7 ((Implementing-Pointer :S2 :N2) :sit)))
(Assert '((History :S1 :S2) :sit)
 '(implementation :f1 :f2 :f3 :f4 :f5 :f6 :f7)))
```

The REPRESENT clauses in IMPLEMENTATION-METHODS are handled almost exactly like RELATION-DEFINITIONS. In the next chapter, I will discuss how REASON analyzes side-effects, discussing the problems posed by defined relations extensively at that point. The reader should bear in mind that by defined relations I mean both RELATION-DEFINITIONS and the equivalences given in the REPRESENT clauses.

Before moving on to a brief catalogue of object-type descriptions, I should observe that the terms abstract object and concrete object used in IMPLEMENTATION-METHODS are a bit of a misnomer. Indeed, sometimes ARRAYS are used to implement other ARRAYS, making the notion of abstract and concrete fuzzy at best. This type of representation is hidden in the above example about STACKS, where we talked about the HISTORY. This is, in fact, nothing but another ARRAY represented as a SUB-ARRAY of the first.

It is a well known technique (used in sorting programs for example) to divide a single ARRAY into SUB-ARRAYS, using indices to separate the conceptually distinct "virtual" arrays. Thus, given an array and two numbers (a and b) which are indices of that array we can represent another array of size $(b - a)$ as follows: The i th element of the virtual array is the $(a + i)$ th element of the concrete array. This is expressed as follows:

```
(Implementation-method Array-Segment
 (Abstract-Object: Array)
 (Concrete-Objects:
  (Implementing-Array (type-restriction:
    (Array (whose: (members-type: [members-type Array])))))
  (Lower-Index (type-restriction: Integer))
  (Upper-Index (type-restriction: Integer))
  (restriction: (> [size Implementing-Array] [size Array])
    (Index Implementing-Array Lower-Index)
    (Index Implementing-Array Upper-Index)
    (> Upper-index Lower-Index)
    (Plus lower-index [Size Array] Upper-Index)))
 (represent: (item array number object))
 (definition: (Item A N Obj)
  (<=> (item Implementing-Array [plus lower-bound N] Obj))))
```


Section 10.4: A Catalogue of Object Descriptions

So far, I have introduced a number of mechanisms for the description of objects. In this section I will present a systematic development of REASON's knowledge about data. First, I will state the basic mathematical knowledge about SETS, MAPPINGS, etc. Then I will define several basic programming data-structures, leading up to LISTS, TREES and ASSOCIATIVE-RETRIEVAL HASH-TABLES like that coded in the scenario.

I will begin by defining SET as an object-type with MEMBERSHIP, UNION, INTERSECTION, etc. relationships.

```
(Object-type-definition Set
  (parameters: Members-type (type-restriction: Object-type)
    Size (type-restriction: Pos-Number))
  (partition: Set into: (Non-Empty-Set Empty-Set)
    (allows: Non-Empty-Set (member)))
  (Relation: (Member Set Members-type))
  (Type-restriction (Member Set) Members-type)
  (Restriction: (Cardinality Set Size))
  (Relation: (Union Set Set Set)
    (definition: (Union S-1 S-2 S-3)
      <=> (Equiv (:e1) (Or (Member S-1 :e1)(Member S-2 :e1))
        (Member S-3 :e1)))))
  (Relation: (Intersection Set Set Set)
    (definition: (Intersection S-1 S-2 S-3)
      <=> (Equiv (:e1) (And (Member S-1 :e1)(Member S-2 :e1))
        (Member S-3 :e1)))))
  (Relation: (Set-Minus Set Set Set)
    (definition: (Set-Minus S-1 S-2 S-3)
      <=> (Equiv (:e1) (And (Member S-1 :e1)(Not (Member S-2 :e1)))
        (Member S-3 :e1)))))
```

188 Description of Data-Structures

```
(Relation: (Equal Set Set)
 (definition: (Equal S-1 S-2)
  <=> (Equiv (:el) (Member S-1 :el)
        (Member S-2 :el))))

(Relation: (Size Set Pos-Number)
 (definition: (Size S N)
  <=> (If (Object-type S Empty-Set)
    then (Size S 0)
    else (For-all (Member S :el)
      (Size [Set-Minus S { :el}]
        [Minus N 1]))))))
```

Notice that in the definition of size, I introduced the braces (`{ ... }`) notation for set presentation. REASON (using the macro character facilities of MacLisp) translates this notation into the form which is used internally. Set presentation with braces can take either of two forms, *extensional* or *intentional*; in the former the set is presented by listing its members within the braces:

```
(1 2 5 9)
```

Intentional presentation defines the members of a set as those objects which satisfy a formula with one free variable. The variable is given before a vertical bar `|` and the sentence afterwards. For example, the set of pigs with wings is presented as follows:

```
(:x| (and (pig :x)(winged :x)))
```

In the internal form, these are represented as follows:

```
(Extensional-Set <List of Objects> Set)
(Intentional-Set :variable Pred Set)
```

for example,

```
(Extensional-Set (1 2 5 9) S-1)
(Intentional-Set :x (and (pig :x)(winged :x)) S-2)
```

which say respectively that s-1 is the set of the numbers 1 2 5 and 9 and that s-2 is the set of pigs with wings.

The following set of REASON rules interpret these assertions:

```
(Rule ((:f (Intentional-Set :var :pred :set)))
  (Assert '(object-type :set Set) '(Int-Set-Type :f))
  (Rule ((:f :pred))
    (Assert '(Member :Set :Var) '(Int-Set-Mem :f :g))))

(Rule ((:f (Intentional-Set :var :pred :set))
  (:g (Not :pred)))
  (Assert '(Not (Member :set :var)) '(Int-Set-Not-Mem :f :g)))

(Rule ((:f (Extensional-Set :list :set)))
  (Assert '(object-type :set Set) '(Ext-Set-Type :f))
  (Mapc '(Lambda (x)
    (Assert '(Member :set x) '(Ext-Set-Mem :f)))
    :list))

(Rule ((:f (show (not (member :set :object)) by (ext-set)
  for :goal in :context))
  (:g (extensional-set :list :set)))
  (Or (member :object :list)
    (Assert '(Not (member :set :object)) '(ext-set-not-mem :g))))

(Rule ((:f (goal (Member :set :obj) for :goal in :context))
  (:g (Intentional-Set :var :pred :set)))
  (Propose-Method
    '(Method :f (Int-Set :g)) '(IntSet :f :g)
    (let ((:new-pred (subst :obj :var :pred)))
      (goal-assert :new-pred '((Member :set :obj) . :goal)
        :context '(Int-set-subgoal :f :g)))))
```

where the above is a consequent rule which is used only when requested. It simply states that a sufficient sub-goal is to show that the object satisfies the defining predicate.

The next mathematical object-type in REASON's knowledge base is MAPPINGS:

```
(Object-type-definition Mapping
  (parameters: domain-type (type-restriction: object-type)
                range-type (type-restriction: object-type))
  (is-a: (Set
    (whose: (members-type: (association
      (whose: (key-type: domain-type)
              (value-type: range-type)))))))
  (Relation: (Image Mapping domain-type range-type)
    (definition: (Image Map Key Value)
      (<=> (There-is (:el) (Member Map :el)
        such-that (And (Key :el Key)(Value :el Value))))))
  (Relation: (Domain Mapping (Set (whose: (Members-type: domain-type))))
    (definition: (Domain Map D)
      (<=> (Id D (:el) (There-is (:as) (Member Map :as)
        such-that (Key :as :el))))))
  (Relation: (Range Mapping (set (whose: (members-type range-type))))
    (definition: (Range Map R)
      (<=> (Id R (:el) (There-is (:as) (Member Map :as)
        such-that (Value :as :el))))))
  (Relation: (Range-Element Mapping Range-type)
    (definition: (Range-element Map R)
      (<=> (Member [Range Map] R))))
  (Relation: (domain-Element Mapping domain-type)
    (definition: (domain-element Map R)
      (<=> (Member [domain Map] R))))
```

Notice that I used the notion of an ASSOCIATION (or PAIR) in defining a MAPPING. The following defines ASSOCIATIONS:

```
(Object-type-definition Association
  (parameters: Key-type (type-restriction: object-type)
                Value-type (type-restriction: object-type))
  (parts: Key (type-restriction: Key-type)
           Value (type-restriction: Value-type)))
```

I will now define two more specific kinds of MAPPINGS, namely FUNCTIONS and 1-TO-1 MAPPINGS.

```
(Object-type-definition Function
  (Defining-restriction: (subtype function mapping)
    (restriction: (For-all (:d) (domain-element Mapping :d)
      (there-is-a-unique (:as) (Member Mapping :as)
        such-that (key :as :d))))))
```

```
(Object-type-definition 1-to-1-Mapping
  (defining-restriction: (sub-type 1-to-1-mapping function)
    (restriction: (For-all (:r) (Range-element function :r)
      (there-is-a-unique (:as) (member function :as)
        such-that (value :as :r))))))
```

Notice that since FUNCTIONS are MAPPINGS and 1-TO-1-MAPPINGS are FUNCTIONS, each of these may be invoked with the DOMAIN-TYPE and RANGE-TYPE parameters specified in the MAPPING definition.

A SEQUENCE is a mapping from a NUMERIC-INTERVAL to a SET:

```
(Object-type-definition: Sequence
  (Parameters: Size (type-restriction: Pos-Integer)
    Members-type (type-restriction: Object-type))
  (is-a: (Function
    (whose: (domain:
      (Numeric-Interval
        (whose: (lower-limit: 0)
          (upper-limit: size))))
    (range-type: Members-type)))
  (rename: (Range-element to: Member)
    (domain-element to: Index))))
```

Notice that I used a *rename* clause within the IS-A expression above. This simply means that what was called RANGE-ELEMENT in the type FUNCTION is called MEMBER in the type SEQUENCE. REASON copies in the old definition, substituting MEMBER for RANGE-ELEMENT.

The description of SEQUENCES makes use of the notion of a NUMERIC-INTERVAL which has already been given in the text; I will repeat it here as well:

For Complex Program Understanding

```

(Object-definition Numeric-Interval
  (Parameters: (Upper-bound (type-restriction: Integer)
                      Lower-bound (type-restriction: Integer))
    (restriction: (Less-than Lower-bound Upper-bound)))
  (Is-a: (Set (whose: (Members-type: Integers))))
  (definition: (Numeric-Interval (whose: (upper-bound: U)(lower-bound: L)))
    = (:el) (And (less-than :el U)(greater-than :el L))))

```

A SEQUENCE of particular importance (particularly in describing ARRAYS) is a SEQUENCE of positive integers of a given size. Since I will use such objects in defining ARRAYS I will need to define a notion of one such SEQUENCE being within the bounds established by another such SEQUENCE (e.g. a SEQUENCE of indices being within the ARRAY bounds).

```

(Object-type-definition Sequence-of-pos-integers
  (Parameters: Size (type-restriction: Pos-Integer))
  (Is-a: (Sequence (whose: (Members-type: Pos-Integer)
                          (Size: size)))
    (rename: Image to: Item))
  (Relation: (In-bounds Sequence-of-Pos-integers Sequence-of-Pos-integers)
    (definition: (In-bounds S-1 S-2)
      (<=> (For-all (:index) (Index S-1 :Index)
        (Less-than [Item S-1 :Index] [Item S-2 :Index])))))

```

I will now define ARRAYS as they are found in MacLisp (all dimensions are positive integers).

```

(Object-type-definition Lisp-Array
  (Parameters: dimension (type-restriction: Pos-Integer)
    Upper-bound (type-restriction:
      (Sequence-of-Pos-Integers (whose: (size: dimension))))
    Members-type (Type-restriction: object-type))
  (Indexed-part: Item (type-restriction: Members-type)
    (Index-restriction: Index
      (type-restriction: (Sequence-of-Pos-Integers (whose: (size: dimension))))
      (restriction: (Within-bounds Index Upper-bound))))
  (Relation: (Index-of Lisp-Array
    (Sequence-of-Pos-Integers (whose: (size: dimension))))
    (definition: (Index-of Array Seq)
      (<=> (Within-bounds Seq Upper-bound))))

```


I will now move to a more complicated data-structure, namely hash-coded data-bases. There are several such systems, the features common to all of them is the use of a HASHING-FUNCTION and an ARRAY. I will start by defining a HASH function.

```
(object-type-definition hash
 (parameters: domain-type (type-restriction: object-type)
               size (type-restriction: pos-integer))
 (is-a: function (whose: (domain-type: domain-type)
                        (range: (numeric-interval:
                                (whose: (lower-bound: 0)
                                      (upper-bound: size)))))))
```

The simplest hashing system is one which calculates a HASH from the entire data-base item. The next simplest is an associative system which calculates the HASH on the KEY part of the data-base item. Both these insert the item into a single place in the data-base. A more complicated system will be described later.

```
(Object-type-definition hashtable
 (Parameters: Members-type (type-restriction: object-type)
               size (type-restriction: pos-integer)
               hash (type-restriction: (hash
                                         (whose: (domain-type: Members-type)
                                               (size: size))))))
 (is-a: (lisp-array (whose: (members-type:
                             (set: (whose: (members-type: Members-type)))
                             (size: size)))
                    (rename: item to: bucket-part))
 (Relation: (Member hashtable Members-type)
 (definition: (Member ht el)
               (> (Member [bucket-part ht [hash el]] el))))
```

Now for the associative version of HASH-TABLES:

For Complex Program Understanding

```

(Object-type-definition associative-hashtable
  (Parameters: key-type (type-restriction: object-type)
               value-type (type-restriction: object-type)
               size (type-restriction: pos-integer)
               hash (type-restriction:
                     (hash (whose: (domain-type: Key-type)
                                (size: size))))))
  (is-a:
    (lisp-array
      (whose: (members-type:
                (set: (whose: (members-type:
                              (association (whose: (key-type: key-type)
                                                    (value-type: value-type))))))
                  (size: size)))
      (rename: item to: bucket-part))
    (Relation: (Member hashtable Members-type)
      (definition: (Member ht el)
                    (<=> (Member [bucket-part ht [hash [key el]]] el))))))

```

So far I have not dealt with recursively defined structures such as **LISTS**, **TREES**, **GRAPHS**, etc. I will develop these by first defining an object-type called **RECURSIVE-STRUCTURES**. I will then define **LISTS**, **TREES**, etc. as special cases of this object-type.

```

(Object-type-definition Recursive-structure
  (partition: Recursive-Structure into: (terminal non-terminal)
    (allows: non-terminal (immediate-children))))

::: a recursive structure is always built from the "recurring parts" which are
::: called the non-terminals and the "stopping parts" which are called terminals.

(parameters: value-names (type-restriction: set))
(parts: Immediate-Children (type-restriction:
  (set: (whose:
    (members-type: Recursive-structure)))))
Values (type-restriction: (association: (whose: (domain: value-names)))))

::: the definition is parameterized by a set of "values" which are other fields
::: present at each node, but which are not involved in the recursion

::: Now make the basic definitions. The immediate children are the first
::: level of recursion, i.e. the nodes pointed to directly by the current node
(relation: (immediate-child recursive-structure recursive-structure)
  (definition: (immediate-child rs-1 rs-2)
    (> (Member [immediate-children rs-1] rs-2)))

::: A proper node is one gotten to via an immediate child link.
(relation: (proper-node recursive-structure recursive-structure)
  (definition: (Proper-node rs-1 rs-2)
    (> (Or (immediate-child rs-1 rs-2)
      (there-is (immediate-child rs-1 :ic)
        such-that (proper-node :ic rs-2)))))

::: Node is the transitive closure of immediate child. I.e. anything you can
::: get to by first going to an immediate child and then its immediate child, etc.
(relation: (node recursive-structure recursive-structure)
  (definition: (node rs-1 rs-2)
    (> (or (proper-node rs-1 rs-2)
      (id rs-1 rs-2)))))

```


196 Description of Data-Structures

```

::: A terminal node is a node which is a terminal. It stops the recursion.
(relation: (terminal-node recursive-structure recursive-structure)
 (definition: (terminal-node rs-1 rs-2)
  (<=> (and (node rs-1 rs-2)(object-type rs-2 terminal)))))

::: Non-terminal nodes are the other guys.
(relation: (non-terminal-node recursive-structure recursive-structure)
 (definition: (Non-terminal-node rs-1 rs-2)
  (<=> (and (node rs-1 rs-2)(Object-type rs-2 non-terminal)))))

::: If all the non-terminal nodes have the same number of immediate children,
::: this number is called the node degree. Lists have node degree 1, binary
::: trees have node degree 2; some graphs have no degree by this def.
(relation: (Node-degree Recursive-structure Pos-integer)
 (definition: (Node-degree RS N)
  (<=> (for-all (Non-terminal-Node RS :Node)
   (Size [Immediate-Children :Node] N)))))

::: two structures share if they have a common node. This is very important
::: for reasoning about side-effects.
(relation: (shares-structure recursive-structure recursive-structure)
 (definition: (shares-structure rs-1 rs-2)
  (<=> (There-is (proper-node rs-1 :node)
   such-that (proper-node rs-2 :node)))))

::: if you can find a node somewhere in this structure which is a node of itself
::: in a non-trivial way (nodes were defined to be nodes of themselves) then the
::: structure has cycles. This is usually very bad.
(relation: (has-cycles recursive-structure)
 (definition: (has-cycles rs-1)
  (<=> (There-is (node rs-1 :node)
   such-that (proper-node :node :node)))))

::: Structures can be divided into those with cycles and those without
::: notice that structures now have two different partitions.
(partition: recursive-structure into: (cyclic-structure acyclic-structure)
 (dividing-criterion: (Has-cycles cyclic-structure)))

```

```

::: The rule of structural induction can be applied to any acyclic structure
::: to prove that some property holds for all of its nodes.
(proof-rule:
  (goal: Property
    (where: (occurs-in Property :var)
      (object-type :var acyclic-structure)))
  (subgoals: (for-all (:term)(terminal-node acyclic-structure :term)
    .(subst :term :var property))
    (for-all (:non-term) (non-terminal-node acyclic-structure :non-term)
      (implies
        (for-all (:child)(immediate-child :non-term :child)
          .(subst :child :var property))
        .(subst :non-term :var property))))))

```

This is the only object-type definition so far where I have found it useful to include a PROOF-RULE with the definition. The syntax is, therefore, somewhat *ad hoc*. REASON builds a consequent reasoning rule and a method-proposer from this statement. The method-proposer will trigger if there is a goal statement which includes an object which is an ACYCLIC-STRUCTURE. If the method is accepted, it creates the two sub-goals of showing that (1) The property holds for all TERMINAL-NODES of the object and (2) If it holds for all IMMEDIATE-CHILDREN of a NODE then it holds for the NODE itself. Notice the use of commas (,) in front of the SUBST expressions to indicate that they should be evaluated (i.e. SUBST is to be invoked as a function; it is not a predicate name). STRUCTURAL-INDUCTION is also used on BINARY-TREES and LISTS, but the rule is simply copied into their definitions since they are defined as special kinds of RECURSIVE-STRUCTURES. [Boyer & Moore, 1975,77] uses structural induction extensively to prove theorems in recursive function theory and Pure Lisp.

Next I will define a BINARY-TREE as a special kind of ACYCLIC-RECURSIVE-STRUCTURE whose IMMEDIATE-CHILDREN is a set of two BINARY-TREES.

```

(Object-type-definition Binary-tree
  (Parts: Left (type-restriction: binary-tree)
    Right (type-restriction: binary-tree))
  (Is-a: (Acyclic-structure (whose: (Node-degree: 2))))
  (map: (Immediate-children binary-tree)
    into: ([left binary-tree][right binary-tree])))

```

Notice the use of the *map* clause above. Remember that whenever a object is defined with an *is-a* clause the system copies all the information about the super-type into the new definition. The *map* clause tells REASON that any expression involving IMMEDIATE-CHILDREN in the definition of RECURSIVE-STRUCTURES should be replaced by an identical expression involving the extensionally presented set above. This can be regarded as an equivalence saying that if a BINARY-TREE is thought of as a RECURSIVE-STRUCTURE then the set of IMMEDIATE-CHILDREN is the set composed of the LEFT of the tree and the RIGHT of the tree. In a BINARY-TREE the IMMEDIATE-CHILDREN PART is a virtual object composed of the two "real" parts, the LEFT and the RIGHT.

Finally, I will define a LIST as a data-type with two parts: a FIRST and a REST. The chain of RESTS forms a RECURSIVE-STRUCTURE. There is a substantial private vocabulary associated with LISTS, which is indicated by the *rename* clauses. As above, the *map* clauses are used to indicate how this structure satisfies the definition of RECURSIVE-STRUCTURES. A new construct, the *singleton type* is introduced to take care of the fact that in LISP the only possible EMPTY-LIST is the special object NIL. A SINGLETON-TYPE consists of one object; therefore, any object known to be of that type is also known to be the unique object of that type. Finally, several relations peculiar to LISTS are introduced.


```

(Object-type-definition List
  (Parameters: Members-type (type-restriction: Object-type))
  (Parts: First Rest (type-restriction: List))
  (partition: List into: (Empty-list Non-Empty-list)
    (allows: Non-Empty-List (first rest member)))
  (singleton-type: Empty-list object: Nil)
  (is-a: Recursive-structure (whose: (Node-degree: 1))
    (map: ((immediate-children list) into: ([rest list]))
      ((values list) into: ((first . [first list])))))
  (rename: (node to: sublist)
    (terminal to: empty-list)
    (non-terminal to: non-empty-list)
    (non-terminal-node to: non-empty-sublist)
    (terminal-node to: empty-sublist)))
  (relation: (Member List Members-type)
    (definition: (Member L O)
      (<=> (Or (First L O)
        (Member [rest L] O))))))
  (function: (Length List Pos-Integer)
    (definition: (Length L N)
      (<=> (if (Object-type L Empty-list)
        then (id N 0)
        else (Length [rest list] [Minus N 1] )))))
  (relation: (Comes-before-in List Members-type Members-type)
    (definition: (Comes-before-in L O-1 O-2)
      (<=> (There-is (Sublist L :Sub)
        Such-That (And (First :Sub O-1)
          (Member [Rest :Sub] O-2))))))

```

Notice that in the `MAP` clause, the `VALUES` part of the `RECURSIVE-STRUCTURE` is mapped onto the pairing of the name `"FIRST"` with the `FIRST` part of the `LIST`.

The final object I will present is the `CONNIVER`-style associative-retrieval hashing system. This is an `ARRAY` each of whose items are `LISTS`. The array holds `ASSERTIONS` which are `BINARY-TREES`. An `ASSERTION` is a `MEMBER` of the `TABLE` if it is a `MEMBER` of each `BUCKET` HASHED to by any of its `TERMINAL-NODES`.

For Complex Program Understanding

```

(Object-type-definition Conniver-hash-table
  (Parameters: size (type-restriction: pos-integer)
               hash (type-restriction:
                    {hash
                      (whose:
                        (domain-type: (pair: (whose: (left: Atom)
                                                    (right: Pos-Number))))
                        (size: size))))))
  (is-a: (lisp-array (whose: (members-type:
                               (set: (whose: (members-type: binary-tree-of-atoms))))
                      (size: size)))
         (rename: item to: bucket-part))
  (Relation: (Member hashtable Binary-Tree-of-Atoms)
  (definition: (Member ht el)
    (<=> (for-all (:node) (terminal-node el :node)
                (for-all (:pos) (position-in ht :node :index)
                    (Member [bucket-part ht [hash :node :index]] el))))))

```

The full-blown programmer's apprentice system will have an even more extensive catalogue of descriptions; however, my point here is not so much to present the complete catalogue used in REASON, as to show how the reasoning system gets its information. The complete catalogue used by the apprentice is being worked on by Rich [Rich, 1977,78]. It is also for this reason that I did not include spec-types for the various operations associated with object-types as is done in data-abstraction languages. The final catalogue will do so, but for the reasoning system's purposes this is not necessary.

Chapter 11: Reasoning About Side-effects

The ability to change the structure of an object while leaving its identity unchanged provides a powerful mechanism for modularity and abstraction in advanced programming languages such as LISP. However, precisely because side-effects on global and shared structures possess such potential power, they also allow enormous room for error. When a segment causes a side-effect it saves itself the worry of communicating with a hoard of other segments which might access the same structure, but it does so at the price of requiring an assurance that it is safe to make the proposed change. This assurance, unfortunately can only be gained by engaging in a non-local and expensive form of reasoning.

Simple changes can result in non-trivial results, something every experienced programmer has learned the hard way. Since complex structures are built from less complex objects, it follows that a side-effect to a part of a structure can change properties of the whole structure. Even worse, since complex structures may share sub-structure, a modification to one data structure might change a property of some other data structure which had been thought of as a completely separate object. In reasoning about the results of a particular action, REASON must assess what properties besides those explicitly stated will also change.

In the context of common sense reasoning in AI this general problem has been termed the frame problem in [McCarthy and Hayes, 1967 & 69] and has received a considerable degree of attention [Raphael, 1970], [Hayes, 1971a & b]. An example will illustrate the problem. Suppose I tell you that the saucer was taken to the kitchen. If you knew that the cup was on the saucer, then you would probably infer that the cup was now in the kitchen; the inference would probably be correct since there is a causal relationship between the location of the cup and the location of the saucer upon which it is placed. However, you would also never think to ask whether the saucer had changed color when it was removed to the kitchen, since motion has little to do with position.

In common sense reasoning, one has to assume that most things don't change unless there is strong reason to believe that they do. When such assumptions lead to trouble, one re-examines his current belief system and rearranges things to correspond to the realities.¹ In the case of program understanding, similar techniques also apply. Consider the following procedure for swapping the first element of two lists of numbers without using a temporary variable:

For Complex Program Understanding


```

(defun swap (list-1 list-2)
  (replace list-1 (- (car list-1)(car list-2)))
  (replace list-2 (+ (car list-1)(car list-2)))
  (replace list-1 (- (car list-2)(car list-1))))

```

a brief explanation of this procedure might be helpful. Suppose that the CAR of LIST-1 is A and the CAR of LIST-2 is B. Then the sequence of additions and subtractions leaves the following values in the first position of the two lists:

	list-1	list-2	reason
initially	a	b	
first sub	a - b	b	
addition	a - b	a	$b + (a - b) = a$
2nd sub	b	a	$a - (a - b) = b$

Interestingly enough, this program has a bug; furthermore, few programmers (even experts) spot the bug when examining the program. (The reader might try to figure out what the problem is now before proceeding).

The problem is illustrated by considering what this program will do if called with the same object for both arguments:

```

(swap list-23 list-23)

```

Since the formal parameters LIST-1 and LIST-2 are bound to the same object, the procedure fails, putting 0 into the CAR of LIST-23. The fact that most programmers fail to spot this bug indicates that they are assuming that the two arguments are distinct lists, even though they have no evidence supporting this assumption.

This indicates that, as in common sense reasoning, programmers use more than one strategy for analyzing the effect of an action. In the more reckless strategy, one assumes that things are not changed unless there is reason to believe they do. This has the advantage of being right most of the time, requiring less effort, and allowing the programmer to form a "first order" theory of what the code does. In the more careful strategy, one does the opposite, assuming things are affected unless evidence to

the contrary exists. This, has the advantage of never allowing a false conclusion to be drawn, but the disadvantage of requiring much greater effort, to the extent that it can prevent one from forming a "first order" understanding. In developing REASON, I have experimented with two protocols corresponding to these two forms of analysis. I will present these protocols after presenting some notational preliminaries in this next section.

Section 11.1: Specifying Side-effects

REASON allows special kinds of spec clauses to facilitate the description of side-effects. The two such basic clauses are `SIDE-EFFECT` and `NEW` which have the following format:

```
(Side-Effect changed-object Input-Sit Output-Sit New-clause)
(New New-Object Input-Sit Output-Sit New-Clause)
```

The first of these states that on the transition from the input situation of the segment to the output situation of the segment the `CHANGED-OBJECT` is subjected to a side-effect which makes `NEW-CLAUSE` true in the output situation. The second type of assertion states that a new object is brought into existence (through use of the `CONS` or related operations) on the transition from the input to the output situation of the segment; the new object satisfies the property stated in `NEW-CLAUSE`.

Within a segment's specs clauses the transition part of these statements is omitted since it can unambiguously be inferred by the symbolic interpreter. Thus, in specs one may write

```
(Side-effect object clause)
(New object clause)
```

Since a side-effect changes properties of an object we would like a simple way to talk about the object both before and after the side-effect is performed. We already have one mechanism, the object state description (`<object situation>`) which allows us to distinguish between input and output states of the object. However, for notational simplicity we also introduce another method. We allow the `OUTPUTS` clause to create a

For Complex Program Understanding

second name for an input object, as follows:

```
(outputs: (new-name id-to input-name))
```

This new name is referred to as the *output name*, whenever it is used in an **ASSERT** clause, it implicitly refers to the object in the output situation. Notice that **spec** clauses are usually stated without explicit situation tags since these can be provided by the symbolic interpreter using simple defaulting rules.

The specs for a simple side-effect like **RPLACD** can be stated as follows using this notation:

```
(defspecs rplacd
  (inputs: a-list new-rest)
  (expect: (object-type a-list list))
  (outputs: (the-new-list id-to a-list))
  (assert: (side-effect a-list (rest the-new-list new-rest))))
```

In evaluating a set of specs, the symbolic interpreter builds a mapping which matches input and output ports to objects. An output port mentioned in an **id-to** clause is bound to the same object as is the input port of the clause. In the course of interpreting the **spec** clause, the interpreter replaces input and output ports by the objects to which they are matched. Also the interpreter examines whether there are any output port names in the clause; if so, and if the clause is not explicitly tagged with a situation name, the output situation is added in. Thus, assuming that **RPLACD** were applied to **LIST-1** and **REST-1** in **S-0**, resulting in the new situation **S-1**, the above **SIDE-EFFECT** clause would actually be asserted as:

```
(side-effect list-1 S-0 S-1 (rest list-1 rest-1))
```

This defaulting mechanism is quite useful in describing side-effects which relate some property which is true of the object on input to some property true upon exit. For example, we might want to increment by 1 some count field of a particular data-structure. We would then say that the count field in the output situation is 1 plus the count field in the input situation. This, of course, involves a reference expression; the same defaulting rules apply here, if the reference expression mentions

no output port it is defaulted to the input situation.

```
(defspecs bump
  (inputs: a-record)
  (expect: ... )
  (outputs: (the-new-record id-to a-record))
  (assert:
    (side-effect a-record
      (count the-new-record [plus 1 [count a-record]]))))
```

Assuming that the segment is applied to RECORD-1 in S-0 yielding S-1 this is asserted as:

```
(side-effect record-1 S-0 S-1
  (count record-1 ([plus 1 ([count a-record] S-0) ] S-0)))
```

Notice that in making the defaults a reference expression which is resolved in the input situation is regarded as an input name, while one resolved in the output situation is regarded as an output name and will force any enclosing expression to be regarded as an output expression. This is not always convenient but it can always be over-ridden by use of explicit object-state descriptions or fully spelled out reference expressions. For example, the LISP function `NREVERSE` may be described as follows:

```
(defspecs nreverse
  (inputs: a-list)
  (expect: (object-type a-list list))
  (outputs: new-list)
  (assert: (((last-cell a-list newlist) *before*)
    (reverse <a-list *before*> <new-list *after*>))))
```

where, as mentioned in Chapter 5, `*BEFORE*` and `*AFTER*` are special symbols provided by the symbolic evaluator to stand for the input and output situations of the segment. These specs state that (1) the output list was the last cell of the input list at the time of invocation of the segment and (2) that the output list at the time of output has a structure which is the reverse of that possessed by the input list at the time of input.

For Complex Program Understanding

It should be noted that although the `SIDE-EFFECT` and `NEW` clauses are special in the sense that they are explicitly trans-situational assertions, they are otherwise normal. In particular, `SIDE-EFFECT` clauses can be part of a quantified statement. For example, we might want to say that certain members of a set (those having a particular `KEY`) were side-effected to turn on a particular `MARK`. (This is done in one version of a `FAST-INTERSECTION` routine). This can be expressed as follows:

```
(defspecs mark-some
  (inputs: the-list the-key)
  (expect: (object-type the-list list)
            (object-type the-key key))
  (outputs: (the-new-list id-to the-list))
  (assert:
    (for-all (:member) (member the-new-list :member)
      (implies (key :member the-key)
        (side-effect :member (marked :member))))))
```

This quantified statement is treated in exactly the same manner as other statements; it is asserted after the objects have been substituted for the input and output port names. Asserting a `FOR-ALL` statement, as we have seen in Chapter 4, creates a rule which triggers if its pattern is matched. This, in turn will create a rule for the `IMPLIES` statement, which triggers if its pattern is matched. If both these patterns are matched, (which is equivalent to saying that we know about a particular object which both is a member of the list and has the appropriate key) then the system will conclude that a side-effect definitely took place, namely that this member of the list was marked.

Section 11.2: Reasoning About Simple Side Effects

Reasoning about side effects is conceptualized by thinking of the segment as forming a *transition* between its input and output situations. Side-effect processing consists of deducing which properties can be moved across this transition safely; I, therefore, refer to this process as *transition analysis*. My general approach of explicitly recording dependencies suggests that `REASON` should provide a justification whenever it decides to move a fact across a transition. Similarly, if it decides not to move a fact it should justify this decision as well. These recorded dependencies allow `REASON` to make an initial decision based on a cursory analysis of the circumstances, while still reserving for itself the option of reconsidering in more detail at a later

time.

The basic protocol is as follows: An assertion in the input situation of a transition can be moved to the output situation of the transition only if there is an explicit assertion declaring it *safe* to do so. In phase one of the analysis, simple rules are run to find reasons for not moving a fact. If such cause is found, these rules assert that it is *unsafe* to move the fact. At the end of this process each assertion is asserted to be safe to move; however, the justification for this safe assertion is non-monotonic, depending on the *outriness* of the corresponding unsafe assertion. Thus, if any reason for considering the assertion unsafe had been found, the unsafe assertion will be *in*, causing the safe assertion to be *out*. If at the end of this process a safe assertion is *in* for a particular fact, then the fact will be asserted in the output situation with its justification pointing to the safe assertion. The following rules carry out these operations:

The Default Assumer for the Careful Protocol

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact))
      (:g (:old-fact :in-sit))
      (:h (transition :in-sit :out-sit)))
      (assume '(not (safe :g :in-sit :out-sit))
              '(safety-first :f :g :h)))
```

The Default Assumer for the Fast and Dirty Protocol

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact))
      (:g (:old-fact :in-sit))
      (:h (transition :in-sit :out-sit)))
      (assume '(safe :g :in-sit :out-sit)
              '(reckless-abandon :f :g :h)))
```

The Safe Fact Mover

```
(rule ((:f (safe :old-fact :in-sit :out-sit))
      (:old-fact (:fact :in-sit))
      (:g (transition :in-sit :out-sit)))
      (Assert '(:fact :out-sit) '(safe-from-side-effect :f :g)))
```

Make The Side-effect Appear In The New Situation

```
(rule ((:f (side-effect :object :in-sit :out-sit :new-fact)))
      (assert '(:new-fact :out-sit)
              '(side-effects-happen :f)))
```

where the last of these merely asserts that the relation stated in the side-effect assertion is true in the output situation; this is, of course, independent of the safe assertions.

I have not yet shown any rules for determining what is safe and what is not. The simplest such rule is the rule of direct negation which says that a fact which is explicitly negated by a side-effect is unsafe:

```
(rule ((:f (side-effect :obj :in-sit :out-sit (not :fact)))
      (:g (:fact :in-sit)))
      (Assert '(not (safe :g :in-sit :out-sit))
              '(direct-negation :f :g)))
```

Another simple side-effect rule concerns objects with parts. Suppose that a record is modified to change one of its parts to a new value. It follows, that the assertion stating the old value of the affected part of the record is not a safe assertion. The following rule states this fact:

```
(rule ((:f (side-effect :object :in-sit :out-sit
                        (:part-name :object :new-value)))
      (:g ((object-type :object :type) :in-sit))
      (:h (part :type :part-name))
      (:i ((:part-name :object :old-value) :in-sit)))
      (assert '(not (safe :i :in-sit :out-sit))
              '(part-side-effect :f :g :h :i)))
```

A second rule is that a part-replacing side-effect cannot affect a part assertion involving a different part-name.

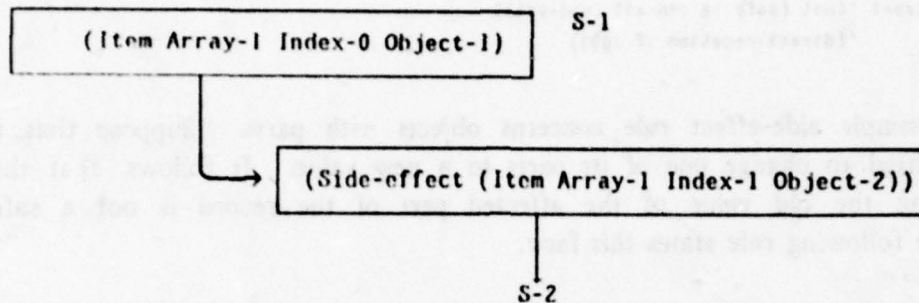
```
(rule ((:f1 (side-effect :object-1 :s-1 :s-2
                        (:new-part-name :object-1 :new-part)))
      (:f2 ((:old-part-name :object-2 :old-part) :s-1)))
      (cond
        ((eq :old-part-name :new-part-name))
        (t
         (assert '(safe :f2 :s-1 :s-2)
                 '(diff-part-side-effect :f1 :f2)))))
```

rules also exists for the independence of indexed-part assertions from part side-effects and vice versa.

The next rule is for side-effects to indexed-parts. This rule introduces a new level of complexity due to the presence of incomplete knowledge. Suppose we have an object with an indexed structure (for example, an ARRAY, a HASH-TABLE, or a record structure including an ARRAY) and that this object is modified changing the part indexed by INDEX-0. Also suppose that we have an assertion saying what is the part indexed by

210 Reasoning About Side-effects

INDEX-1.



Then, the side-effect should make the assertion unsafe if the two indices are identical and leave it unaffected otherwise, as expressed in the following rule:

```

(rule ((if (side-effect :object :in-sit :out-sit
                      (:indexed-part-name :object :new-index :new-value)))
      (:g ((object-type :object :type) :in-sit))
      (:h (:indexed-part :type :index-part-name))
      (:i ((:indexed-part-name :object :old-index :old-value) :in-sit)))
      (assert '(if ((equal :old-index :new-index) :input-sit)
                    then (not (safe :i :in-sit :out-sit))
                    else (safe :i :in-sit :out-sit))
              '(indexed-part-side-effect if :g :h :i)))
  
```

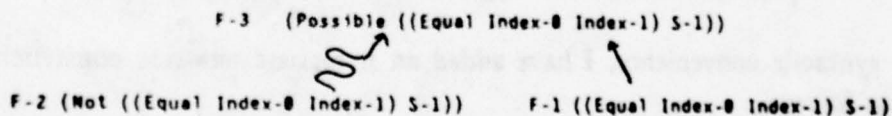
This creates an IF-THEN-ELSE assertion whose justification points to the statements relating to the side-effect. If the premise of the IF-THEN-ELSE (the equality of the two indices) is determined to be true (false), then the THEN (ELSE) clause of the IF-THEN-ELSE is asserted. Its justification includes the IF-THEN-ELSE assertion and the EQUAL assertion or its negation.

Notice, however, that it is altogether possible that neither the premise of the IF-THEN-ELSE, nor its negation are present in the data base and thus, that neither a SAFE nor an UNSAFE assertion will be created. Later in this chapter we will see an example where, due to hypothetical reasoning, it is not possible to know whether the two indices are equal or not, since one of them is an anonymous object, standing for a "typical" index of the array.

This has a very important impact on the protocol for side-effect processing. Recall that this process goes through two passes, the first of which is "fast and dirty" (corresponding to what most programmers would notice without explicitly considering "screw-ball" cases). The second pass is more careful and includes an examination of oddball cases of aliasing (like the swap example, shown earlier). The crucial point here is that, in the first pass analysis, we consider a fact *SAFE* unless evidence to the contrary is found.

If we have an indexed-part side-effect as above, and it cannot be determined whether the two indices are equal, then the rule shown above will make no assertion, (i.e. neither a safe not an unsafe assertion). But in the first pass "fast and dirty" analysis, this lack of an unsafe assertion will be taken as grounds for assuming that the assertion is safe; it will, therefore, be moved across the transition. This will not be logically incorrect, since the justifications for the assumption are explicit and can be withdrawn; it is, however, not a very useful strategy. Even for a fast and dirty pass this strategy is a little too dirty. It would be more useful to say that if it is *possible* that the two indices are equal, then we should not consider the assertion *SAFE*, but should rather do a case analysis, considering separately the two possibilities of equality and non-equality.

Since the conclusion that the assertion is not *SAFE* is based on the *possibility* that the indices are equal, we need a way of asserting that this possibility exists; this *possibility assertion* can then be included in the justification. Because the notion of possibility is used quite frequently, I have developed some syntactic mechanisms to facilitate the use of the concept. The starting point is the observation that an assertion is possible as long as its negation is *out*; of course, if the assertion is *in* it is also possible. Thus, the following support structure captures the notion of *possibility*:



212 Reasoning About Side-effects

This structure is created by calling the function `IS-POSSIBLE` with `F-1` as argument. Calling `IS-POSSIBLE` does not make `F-1` *in* nor does it make the negation of `F-1` *out* it simply creates a support structure which says that if `F-1` is *in* or if `F-1`'s negation, `F-2`, is *out* then `F-3` should be *in*. The result of this is that if `F-1` is possible and `IS-POSSIBLE` is called with `F-1` as argument then the assertion `F-3` will be *in*.

Given this, we can extend the rule for indexed-part side-effects to be more cautious by adding the following to its body:

```
(is-possible ((equal :old-index :new-index) :input-sit))
(rule ((:f (possible ((equal :old-index :new-index) :input-sit))))
  (assert
    (not (safe (:indexed-part-name :object :old-index :old-value)
              :in-sit :out-sit))
    (careful-indexed-part :f)))
```

This says that if it is at all possible that the indices are equal, then the indexed-part assertion should not be declared `SAFE`. If `REASON` decides that moving this assertion is important it can try backward chaining rules on the `IF-THEN-ELSE` assertion to create a case analysis. In one of these cases it will assume that the indices are not equal. This will cause the assertion `F-2` (the inequality of the indices) to come *in*, which will cause the possibility assertion `F-3` to go *out* since its only support is `F-2`. But this, in turn, will *out* the `UNSAFE` assertion since it depended on `F-3`. Finally, the `IF-THEN-ELSE` part of the rule shown earlier will trigger, declaring the assertion to be `SAFE`.

In the other half of the case analysis, if `REASON` assumes the indices to be equal the `IF-THEN-ELSE` part of the rule will trigger, leading to the conclusion that the assertion is not `SAFE`. It will still be true that it is possible for the indices to be equal, so `F-1` will stay *in* as will the `NOT SAFE` assertion derived from it. In this case, `REASON` will have two justifications for believing that the assertion is `NOT SAFE`.

As a further syntactic convenience, I have added an `IF-POSSIBLE-THEN-ELSE` construct, which is invoked as follows:

```

(if-possible (:f fact-1)
  then body-1
  else body-2)

```

If `FACT-1` is possible, then `BODY-1` is executed in a binding environment where `:f` is bound to the possibility assertion for `F-1`. If `FACT-1` is impossible (its negation is *in*), then `BODY-2` is executed in a binding environment in which `:f` is bound to the negation of `F-1`. This is actually a macro for the following:

```

(is-possible fact-1)
(rule ((:f fact-1))
  body-1)
(rule ((:f (not fact-1)))
  body-2)

```


Section 11.3: Safe-from and Not Safe-from

There is still a difficulty in the rules as stated. Whereas it is possible for a single rule to determine that an assertion is NOT SAFE by looking at a single side-effect assertion, it is not possible for it to determine that it is SAFE. It can only determine that the particular side-effect being examined doesn't affect the assertion. There might be other side-effects on this transition, however, which do affect it. It is, therefore, necessary to be more specific in the notation, introducing a *safe from* assertion which states that the assertion is unaffected by a particular side-effect. Similarly, the negation of such an assertion would state that the particular side-effect does affect the assertion in question. Thus if we had:

```
F-1 (Side-effect array-1 s-in s-out (Index array-1 index-1 object-2))
F-2 ((Index array-1 index-0 object-1) s-in)
```

we could write:

```
F-4 (Safe-from F-1 F-2)
```

The SAFE assertion originally used above can only be deduced if the old assertion is SAFE from every side-effect on this transition. To make its assumptions explicit, REASON first gathers up all the side-effects on the transition and explicitly records the assumption that these are all the side-effects.

```
F-1 (side-effects-on-transition s-1 s-2 ( ... ))
```

Also a rule is created which triggers if any other side-effect on this transition is noticed; this rule will negate the assumption that all the side-effects have been considered, thus *ouing* the SAFE assertion and forcing a re-evaluation of the safety of the assertion. Finally, a set of conjunctive goals is established to show that the assertion is SAFE from each side-effect on the transition. If these succeed, the old-assertion is asserted to be SAFE. The SAFE assertion is given a justification which points to each of the SAFE-FROM assertions gathered in the conjunctive goals, plus the assertion F-1 above. This guarantees that if anything is changed (i.e. if a new

side-effect is added, or if one is removed) a recalculation of the true dependencies will be conducted.

An example of this use of SAFE-FROM assertions is the following "fast and dirty" rule for PART side-effects which says that a PART assertion of one object is independent of PART side-effects to another object:

```
(rule ((:f1 (side-effect :object-1 :s-in :s-out
                      (:part-name-1 :object-1 :new-value)))
      (:f2 ((:part-name-2 :object-2 :old-value) :s-in))
      (:f3 ((:object-type :object-1 :type-1) :s-in))
      (:f4 ((:object-type :object-2 :type-2) :s-in))
      (:f5 (part :type-1 :part-name-1))
      (:f6 (part :type-2 :part-name-2)))
  (cond
    ((equal :object-1 :object-2))
    (t (assert '(safe-from :f1 :f2)
                '(diff-obj-part-side-eff :f1 :f2 :f3 :f4 :f5 :f6))))))
```

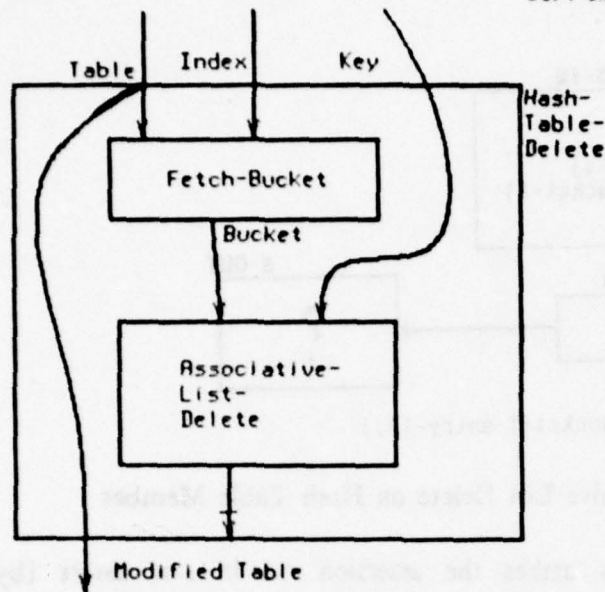
Notice that this is a fast and dirty rule, since even if OBJECT-1 and OBJECT-2 are different object names, it is still possible for them to be anonymous objects which might be identical. The careful version of this same rule, makes this possibility explicit by adding the following:

```
(If-Possible (:g (id :object-1 :object-2))
  then
  (cond
    ((equal :part-name-1 :part-name-2)
     (assert '(not (safe-from :f1 :f2))
              '(poss-id-part-effect :g :f1 :f2 :f3 :f4 :f5 :f6)))
    (t (assert '(safe-from :f1 :f2)
                '(diff-obj-part-side-eff :f1 :f2 :f3 :f4 :f5 :f6))))))
```

Section 11.4: More Complicated Effects

So far the analysis of side-effects has been quite simple considering only assertions about PARTS and INDEXED-PARTS. These are the most primitive notions in the system in the sense that they are not defined in terms of any other programming construct. However, as we saw in our description of programming objects, a host of more complex notions has been developed to allow programs to be thought of in more high level terms.

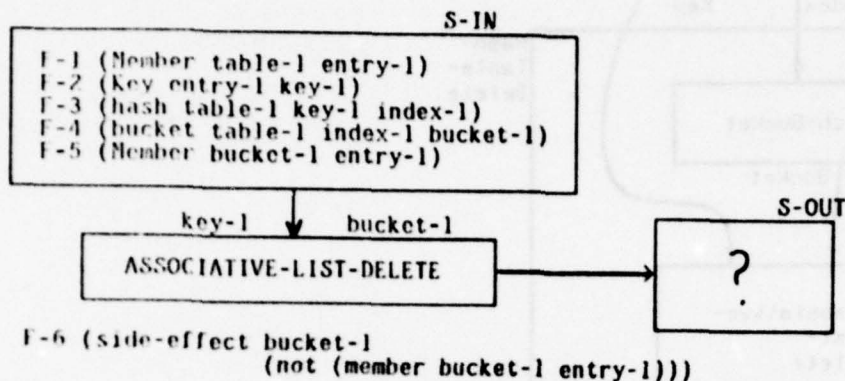
The complex relations which are often used in describing programs are logical combination of assertions which ultimately depend on the PART structure of the objects implementing the more abstract notion. For example, in hashing systems there is a notion of membership in the table which is always defined in terms of membership in one (or more) of the INDEXED-PARTS (BUCKETS) of the table. Similarly, since BUCKETS are frequently implemented as LISTS, membership of an object in the BUCKET reduces to whether the object is the FIRST part or a MEMBER of the REST part of the list. Thus, simple side-effects to the part structure of an object can result in side-effects to derived properties of the object. For example, modifying a TABLE to set one BUCKET to the EMPTY-LIST will (potentially) delete several members of the table. The processes handling side-effects, therefore, must examine the way in which facts in the input situation of a transition depend on one another and use this as a guide to the transition analysis. Consider the following fragment from a HASH-TABLE-DELETE program:



Fragment of A Hash Table Delete Routine

where the LIST-DELETE used here works by side-effect, changing *CDR* pointers so that after its completion, the list will contain exactly those members of its input list which do not have the input key. Suppose that this fragment were part of a larger plan and that in some previous situation of this plan we had concluded that *ENTRY-1* was a member of the *TABLE* since it was a member of *BUCKET-1* which is the bucket hashed to by its key *KEY-1*. Finally, let us suppose that *ENTRY-1* has the same key as that input to the current plan fragment. Obviously, *REASON* ought to conclude that *ENTRY-1* is not a member of the *TABLE* after the *DELETE* operation is performed; let us follow its reasoning process:

It follows from the protocol outlined above that in any transition involving no side-effects, all assertions are safe. Thus, any assertion true at entrance to this fragment will cross the transitions for *FETCH-BUCKET*, reaching *ASSOCIATIVE-LIST-DELETE*. Let us call the input and output situations of *ASSOCIATIVE-LIST-DELETE* *S-IN* and *S-OUT* respectively. We have the following facts:



Effect of Associative List Delete on Hash Table Member

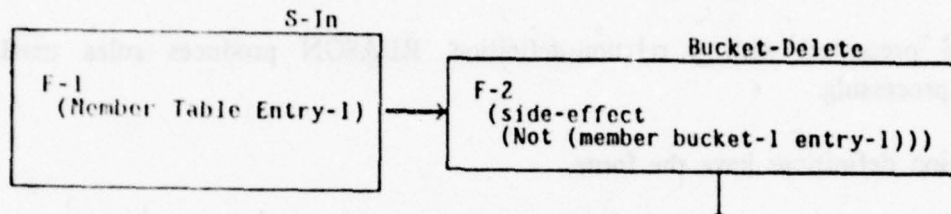
Notice that the side-effect F-6 makes the assertion F-5 in S-IN UNSAFE (by direct negation). We can also use the rules shown so far to determine that F-4 F-3 and F-2 are SAFE from the side-effect F-6; since this is the only side-effect on this transition, these are safe to move across the transition.

The membership assertion F-1 depends on F-2, F-3, F-4 and F-5 since it was derived from these assertions using the relation-definition rule for HASH-TABLE membership. But F-5, one of these facts, is made UNSAFE by the side-effect on this transition. It would seem then that we should follow the justification from F-5 to F-1, concluding that since one of its supports has become UNSAFE, F-1 should also be judged UNSAFE. It would correctly follow that there is no reason to believe that ENTRY-1 is still a MEMBER of the TABLE after the side-effect, i.e. that the side-effect to F-5 has caused a *derived side-effect* to F-1.

This suggests using the justifications to guide an analysis of derived side-effects. It is my feeling that an elegant extension to the TMS dependency system will make this possible (Doyle, McCallester, and Stallman have all suggested this idea in personal communications), however, REASON uses a different method, which is motivated by the fact that the logical connection between facts might not be represented explicitly. In the example above we assumed that we had deduced that ENTRY-1 was a member of TABLE-1, and thus we already had a justification recording what facts this assertion depended on. It was then a simple matter to see that the side-effect which deleted an entry from one of the table's buckets also effected the membership assertion.

However, suppose that this membership assertion had not been deduced, but had only been told to us (as an output assertion of some other sub-segment's specs) or that it had been assumed. Then the only justification for the membership assertion would be a dependency on the spec clause or on the reason for making the assumption.

(spec clause or assumption reason)



Side effect on Unexpanded Defined Relation

Notice that in this circumstance, which is actually much more typical than that shown earlier, there is no set of justifications linking the membership assertion *F-1* to the assertion negated by the side-effect *F-2*. However, consider what would happen if *F-1* were expanded into its definition. This would produce exactly the facts *F-2*, *F-3*, *F-4* and *F-5* which we saw in the earlier example. As these are asserted, they will establish the logical connection between *F-1* and *F-5* that we saw earlier.

The key problem, therefore, is for REASON to identify those circumstances in which it is necessary to force this expansion of defined relations. Rules of the following form would at first glance seem sufficient:

```

(Rule ((:f1 ((Member :table :entry) :s-in))
      (:f2 (Side-effect :table :s-in :s-out
                      (Not (member :bucket :entry))))
      (:f3 ((Object-type :bucket bucket) :s-in)))
(assert (expand ((Member :table :entry) :s-in))
        (expand-for-transition-processing :f1 :f2 :f3)))
  
```

Rules like the one above could be created by analyzing the definition of the relation involved yielding an expansion rule for each clause involved in the definition. However, I have approached the problem somewhat differently. This is discussed in the next section.

For Complex Program Understanding

Section 11.5: Determining What's Affected

As I mentioned earlier defined relations introduce a connection between assertions which must be analyzed in side-effect processing. For example, let us define LIST membership in the standard way: an object is a MEMBER of a LIST if it is either the FIRST of the LIST or a MEMBER of the REST of the LIST. A side-effect changing the FIRST of a LIST might change a membership relation. Similarly, in the above example, we saw the connection between BUCKET parts of a TABLE and membership in the TABLE.

When presented with a relation-definition, REASON produces rules used in transition processing.

Relation definitions have the form:

```
(relation obj1 obj2 ...) <=> <compound-expression obj1 obj2 ... >
```

where the compound expression is a combination formed from the logical connectives AND, OR, NOT, FOR-ALL, THERE-IS, IMPLIES, IF-THEN-ELSE. The compound expression may also involve the use of reference expressions which, in effect, introduce new objects on the right hand side of the definition which are not mentioned in the left hand side. For example:

```
(Member list object) <=> (Or (First List Object)
                               (Member [Rest List] Object))
```

makes reference to the REST of the LIST, which is not an object mentioned on the left hand side. In an expanded form we might write this as:

```
(Or (First List Object)
     (And (Rest List List-1)
           (Member List-1 Object)))
```

where LIST-1 is a new object introduced to resolve the reference expression. From this we can extract two forms of information: One is a network of potential dependency assertions, linking assertions to those side-effects which might make them UNSAFE. The second form of information is a set of REASON rules which assert SAFE and UNSAFE assertions. For example, from the definition for LIST membership we can get the following potential dependency assertions:

```
(potential-dependency (member :list :object-1)
                      (first :list :object-2))

(potential-dependency (member :list :object-1)
                      (rest :list-1 :list-2))

(potential-dependency (member :list :object)
                      (not (member :list-1 :object)))
```

Potential dependency assertions are the information used to determine that there *might* be a logical connection between a fact and a side-effect. These say that if (1) There is an assertion in the input situation which matches the first pattern and (2) There is a side-effect on the transition which matches the second pattern, then it is *possible* that the assertion is rendered UNSAFE by the side-effect. Notice that in the case of dependencies on non-functional relations (such as MEMBER) the dependency is on the negation of the relation. If functional relationships (such as PART OF INDEXED-PART) assertions are involved, a side-effect asserting a new value for the property, such as (FIRST LIST Foo), implicitly negates any previous value of the property, such as (FIRST LIST Bar); for these relations the dependency pattern is not negated. Also note that we have omitted the object-type information that goes with the assertions; however, since these assertions are used only to find things which *might* be affected, omitting the object-type information will simply allow some assertions to be considered even though they are not affected. This can do no harm, it can only make the system overly cautious.

The network of potential dependency assertions is completed by using a transitivity rule to reflect the fact that if F-1 depends on F-2, which in turn depends on F-3, and if F-3 is made UNSAFE by a side-effect, then F-2 and, in turn, F-1 also become suspect:

```
(rule ((:f (potential-dependency :a :b))
      (:g (potential-dependency :b :c)))
      (assert (potential-dependency :a :c)
              (pd-trans :f :g)))
```

The information in the potential dependency assertions is used by a rule which monitors transitions looking for facts which might be made *UNSAFE* by a side-effect. If such situations are noticed, the rule asserts that the fact is *possibly-unsafe*. Any fact which is *POSSIBLY-UNSAFE* is expanded.

```
(rule ((:f (potential-dependency :a :b))
      (:g (side-effect :object :s-in :s-out :b))
      (:h (:a :s-in)))
      (Assert '(possible (not (safe-from :g :h)))
              '(pd :f :g :h)))

(Rule ((:h (Possible (not (safe-from :f :g)))))
      (Assert '(Expand :g) '(pd-expand :h)))
```

The rules for developing the potential dependencies are as follows: If the connective is AND or OR then build a potential dependency for each clause of the conjunction or disjunction. $(\text{IMPLIES } P \ Q)$ is logically equivalent to $(\text{OR } (\text{NOT } P) \ Q)$ and is handled accordingly. Similarly, IF-THEN-ELSE is built from IMPLIES. The quantified statements require a brief explanation. If we have

F-10 (For-all :vars :p :q)

then two kinds of side-effects could make F-10 become not true. One is a side-effect which causes some object which does not satisfy :q to satisfy :p, creating a counter example to the universal quantification. The other is a side-effect to an object which currently satisfies both :p and :q so as to make it no longer satisfy :q. Therefore, universally quantified statements potentially depend on both :p and :q. A similar argument holds for existential quantification.

Two points about these rules for determining potential dependency should be noted. First, these rules only signal the possibility that an assertion is affected by a side-effect; it is for other more thorough rules to explore whether or not the assertion actually is *SAFE* or not. This allows a many layered control structure in which one set of rules notices candidates for examination, and other sets of rules chose to examine these candidates at a level of detail deemed appropriate.

The second point to be made here, is that the potential dependency rules shown so far are actually of the "fast and dirty" variety. Remember that the swap example showed that different local variable names might, in fact, name the identical object. Usually people rule out this possibility of "aliasing" to facilitate their analysis. However, to be completely accurate one must examine all possibilities.

The careful version of a rule for *LIST* membership, for example, is

```
(rule ((:f (side-effect :obj :s-in :s-out (first :obj :new-first)))
      (:g ((member :obj-2 :old-first) :s-in)))
      (if-possible (:h (id :obj :obj-2))
        then (Assert '(possible (not (safe-from :f :g)))
                '(poss-or-se-careful :f :g :h))))
```

This requires that the system have rules for determining whether objects are identical or not, and furthermore that it maintain this information rather carefully. Fortunately, most procedures do not involve a large number of objects so this task is tractable. There are several ways in which the system can deduce the non-identity of objects (we have already discussed ways in which it can determine identity). One rule is that if an object is newly created in a situation which comes after a situation in which a second object was known to exist then the two objects are not identical:

```
(rule ((:f (new :object-1 :s-in :s-out :fact))
      (:g (occurs-in :object-2 (:fact-2 :s-other)))
      (:h (comes-before :s-other :s-out)))
      (assert '(not (id :object-1 :object-2))
              '(diff-date-of-birth :f :g :h)))
```

A second rule for non-identity uses the disjointness relation between types in the object-type hierarchy to infer that two objects have different types and are, therefore, distinct. Finally, both of these are special cases of the general rule that if a property holds of one object but not of the other then those objects are distinct.

Once it has been determined that an assertion is possibly affected by a side-effect it remains to be determined whether the assertion is *SAFE* or *UNSAFE*. A second set of rules is developed from the relation-definitions, by going through the logical connectives used in the definitions. For example, a conjunction in which one conjunct has been side-effected can be deduced to be unsafe. However, a disjunction must be analyzed further. The following rules conduct this analysis for *LIST* membership:

```
(rule ((:f1 (possible (not (safe-from :f2 :f3))))
      (:f2 (side-effect :list :s-in :s-out (first :list :obj-1)))
      (:f3 ((member :list :obj-2) :s-in))
      (:f4 ((Rest :list :rest) :s-in)))
      (if-possible (:f5 ((not (Member :rest :obj-2)) :s-in))
        then (Assert '(Not (safe-from :f2 :f3))
                     '(dj-not-safe :f1 :f2 :f3 :f4 :f5))
        else (Assert '(safe-from :f2 :f3)
                     '(dj-safe :f1 :f2 :f3 :f4 :f5))))
```

If it is possible that the old *FIRST* element of the *LIST* occurred only in the *FIRST* position, then it is possible that the side-effect of changing the *FIRST* of the *LIST* would cause that element to cease to be a *MEMBER* of the *LIST*. Thus, a cautious strategy avoids moving this fact over the transition until more information is known. If it is ever learned that the object was definitely a *MEMBER* of the *REST* of the *LIST* then the assertion will be declared *SAFE* by the second clause of the *IF-POSSIBLE* rule. In the mean time, this cautious strategy prevents any defaulting strategy of the first pass analysis from being too lax.

Notice that side-effect rules such as the one above are triggered by the *possibly unsafe assertion*, rather than by the side-effect assertion directly. (The *POSSIBLY UNSAFE ASSERTIONS* are created by the *POTENTIAL DEPENDENCY* rules). This allows other rules to decide which assertions should be worked on. In a later section we will see a set of rules which rule out possible *UNSAFETY*, helping the system to avoid useless work.

Since we want to consider all side-effects which might arise, let us consider an example in which the above side-effect leads to a *derived side-effect*. Suppose that in addition to the assertions above about LIST membership, we also had an assertion stating that the object deleted from the LIST was a MEMBER of some HASH-TABLE in the input situation. Since we have concluded that the LIST membership assertion was NOT SAFE, if it is POSSIBLE that this LIST is a BUCKET of the HASH-TABLE we should conclude that the HASH-TABLE membership assertion is UNSAFE as well. To start this process, however, we must first state that there has been a side-effect to the LIST so that the POTENTIAL DEPENDENCY rules may trigger. This is done by a simple rule which translates UNSAFE assertions into side-effect assertions:

```
(Rule ((:f (Not (safe-from :f2 :f3)))
      (:f2 (Side-effect :obj :s-in :s-out :se))
      (:f3 (:fact :in-sit)))
      (assert '(Side-effect :obj :s-in :s-out ((Not :fact) :s-in))
              '(trans-se :f)))
```

The effect of this rule is that every time an assertion is determined to be UNSAFE, it is then treated as a side-effect itself, initiating a consideration of derived side-effects. In this case, this will lead to the triggering of the rule for hash table membership:

```
(rule ((:f1 (possible (not (safe-from :f2 :f3))))
      (:f2 (side-effect :list :s-in :s-out (Not (Member :list :obj)))))
      (:f3 ((Member :table :object) :s-in))
      (:g1 ((key :obj :key-1) :s-in))
      (:g2 ((hash :table :key-1 :index-1) :s-in))
      (If-Possible (:h ((bucket :table :index-1 :list) :s-in))
                    then (Assert '(not (safe-from :f2 :f3))
                                  '(se-tab :f1 :f2 :f3 :g1 :g2 :h))
                    else (Assert '(safe-from :f2 :f3)
                                  '(se-tab :f1 :f2 :f3 :g1 :g2 :h))))
```

Thus, the side-effect propagates through the various levels of definition. Notice that when a definition involves reference expressions, these are handled somewhat specially. If the side-effect is to a clause within the definition which has reference expressions inside it, (as does the definition for membership in the LIST which implements a BUCKET of a HASH-TABLE), then these reference expressions are converted to patterns and moved outside the IF-POSSIBLE expression. However, if the side-effect challenges a reference expression nested inside other reference expressions, then the outer references must also be stated in the IF-POSSIBLE construct.

There are still other logical connectives to consider in the building of side-effect rules. Universal quantification presents analytic problems similar to those of disjunction. Consider the definition of being an ALIST, a LIST, all of whose MEMBERS are PAIRS whose LEFT parts are ATOMS:

```
(Object-type list Alist)
(<*) (for-all (:el) (member list :el)
      (object-type [left :el] Atom))
```

As I mentioned above, two different kinds of side-effects can make an assertion of this form UNSAFE. A side-effect which entered a new element into the LIST might challenge such an assertion (if the KEY of the new object isn't an ATOM). Similarly, changing the KEY of one of the existing elements could undo the truth of the quantified statement if the new KEY is not an ATOM. Thus, a side-effect rule for universally quantified statements must trigger in either event and then examine whether to declare the statement UNSAFE. The following rules do this for the above definition:

```
(rule ([:f1 (possible (not (safe-from :f2 :f3)))]
      (:f2 (side-effect :obj :s-in :s-out (key :obj :key-2)))
      (:f3 ((object-type :list alist) :s-out)))
      (if-possible (:f4 ((member :list :obj) :s-in))
        then (assert '(not (safe-from :f2 :f3))
                     '(uq-n-safe-cautious :f1 :f2 :f3 :f3))
        else (assert '(safe-from :f2 :f3)
                     '(uq-safe-1 :f1 :f2 :f3 :f4))))

(rule ([:f1 (possible (not (safe-from :f2 :f3)))]
      (:f2 (Side-effect :obj :s-in :s-out (member :list :obj)))
      (:f3 ((object-type :list alist) :s-in))
      (:f4 ((key :obj :key) :s-in))
      (:f5 ((not (object-type :key atom)) :s-in))
      (if-possible (:f5 ((not (object-type :key atom)) :s-in))
        then (Assert '(Not (safe-from :f2 :f3))
                     '(uq-not-safe-cautious :f1 :f2 :f3 :f4 :f5))
        else (Assert '(safe-from :f2 :f3)
                     '(uq-safe-2 :f1 :f2 :f3 :f4 :f5))))
```

The above rules are examples of the "fast and dirty" type in that they do not attempt to check for the identity of anonymous objects. A second version of these rules (along the lines shown earlier for the "second pass" rules) does the extra checking.

Section 11.6: An Example

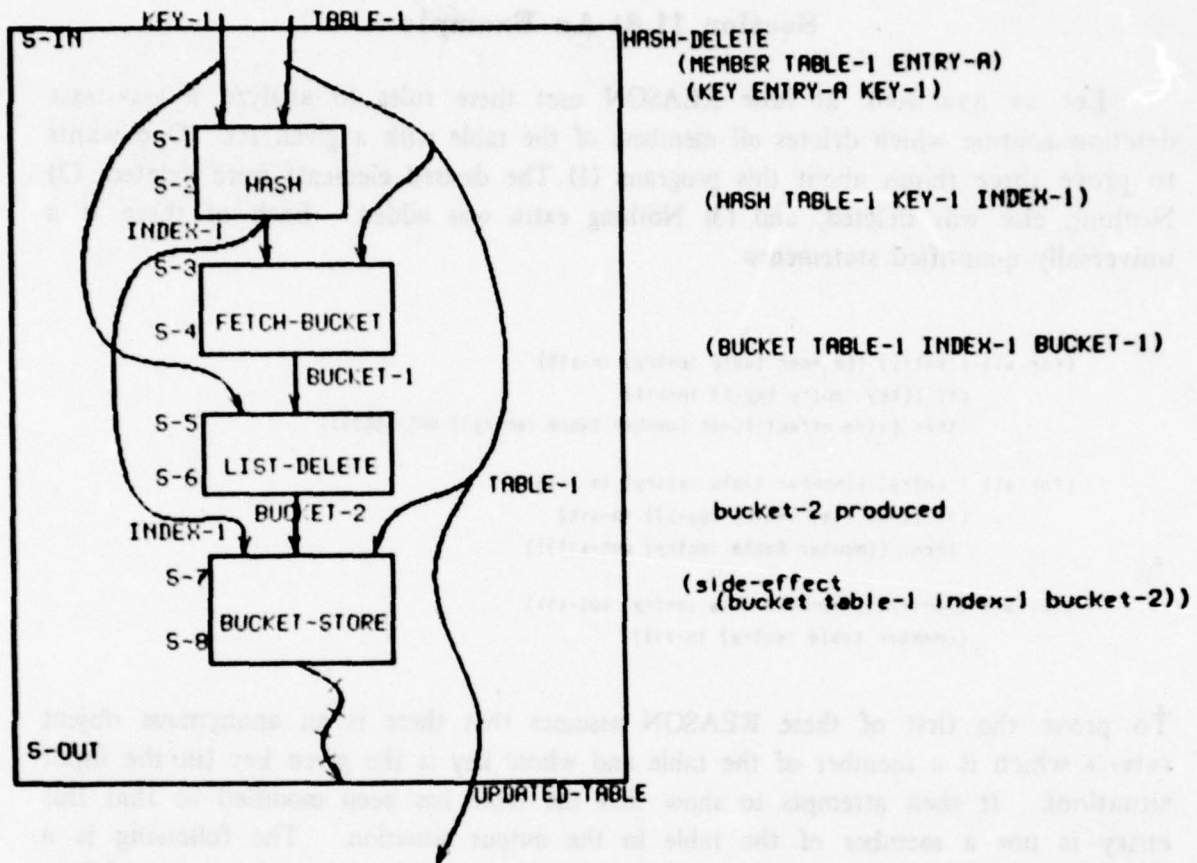
Let us now look at how REASON uses these rules to analyze a HASH-TABLE deletion routine which deletes all members of the table with a given KEY. One wants to prove three things about this program: (1) The desired elements were deleted, (2) Nothing else was deleted, and (3) Nothing extra was added. Each of these is a universally quantified statements:

```
(for-all (:entry) ((member table :entry) in-sit)
  (if ((key :entry key-1) in-sit)
    then (side-effect ((not (member table :entry)) out-sit))))

(for-all (:entry) ((member table :entry) in-sit)
  (if ((not (key :entry key-1)) in-sit)
    then ((member table :entry) out-sit)))

(for-all (:entry) ((member table :entry) out-sit)
  ((member table :entry) in-sit))
```

To prove the first of these REASON assumes that there is an anonymous object ENTRY-A which is a member of the table and whose key is the given key (in the input situation). It then attempts to show that the table has been modified so that this entry is not a member of the table in the output situation. The following is a complete plan diagram for the program with accompanying assertions which follow from the symbolic evaluation.



Plan Diagram for Hash Table Delete

As I described above the membership assertion involving ENTRY-A will pass through each of the first transitions, but will be stopped by the transition representing the action of BUCKET-STORE. The assertion will then be expanded into its definition:

```

((Member Table-1 Entry-A) s-in) <=> ((key Entry-A Key-1) s-in)
                                         ((hash Table-1 Key-A Index-A) s-in)
                                         ((bucket Table-1 Index-1 bucket-1) s-in)
                                         ((Member bucket-1 Entry-A) s-in)
  
```

The support structure for the UNSAFE assertion associated with the membership assertion is now constructed. This structure makes the SAFETY of the MEMBERSHIP assertion depend on the SAFETY of the BUCKET assertion. But since this assertion is UNSAFE, the membership assertion is also deduced to be UNSAFE. However, the assertion


```
((Member bucket-1 Entry-A) s-in)
```

does move safely up to the LIST-DELETE routine whose specs say that it creates a new bucket which contains all entries in BUCKET-1 except those whose key is KEY-1. ENTRY-A, however, is asserted to have KEY-1 as its key; it is, therefore, not a member of BUCKET-2, the output of LIST-DELETE. We have:

```
((Not (member bucket-2 entry-A)) s-6)
```

Notice that this assertion is SAFE to cross the transition representing the BUCKET-STORE side-effect, as are the HASH and KEY assertions. Thus in s-7, the output situation of BUCKET-STORE we have:

```
((Not (member bucket-2 entry-a)) s-7)
((hash table-1 key-1 index-1) s-7)
((key entry-a key-1) s-7)
((bucket table-1 index-1 bucket-2) s-7)
```

from which the antecedent inference rule corresponding to the relation-definition for hash-table membership infers that ENTRY-A is not a member of the table. Since this inference depends directly on a side-effect at this transition it is also a side-effect. We thus have:

```
(Side-effect table-1 s-in s-out
  ((Not (member table-1 entry-1)) s-out))
```

which was the sub-goal needed to deduce the desired universally quantified statement. So we have shown that all entries with the given key are deleted.

Now REASON has to show that nothing was deleted which should not have been. Again it creates an anonymous object ENTRY-B assuming that ENTRY-B is a member of the table and that its key is not KEY-1. The facts propagate similarly to above; however, when REASON tries to expand this assertion it discovers that it does not know the key of ENTRY-B (we only know that its key is not KEY-1) and, therefore, that we also don't know the index this key hashes to or the bucket which is in that slot of the TABLE. Anonymous objects are created to stand in for all of these.

For Complex Program Understanding

```

((Member table-1 Entry-B) s-in) <=> ((key Entry-B Key-B) s-in)
                                     ((Hash table-1 Key-B Index-B) s-in)
                                     ((bucket table-1 Index-B Bucket-B) s-in)
                                     ((Member Bucket-B Entry-B) s-in)

```

The transition processing becomes somewhat more complicated. We have the following assertions and side-effects involved:

```

((bucket table-1 Index-B Bucket-B) s-in)

(Side-effect table-1 s-in s-out
  ((bucket table-1 index-1 bucket-2) s-out))

```

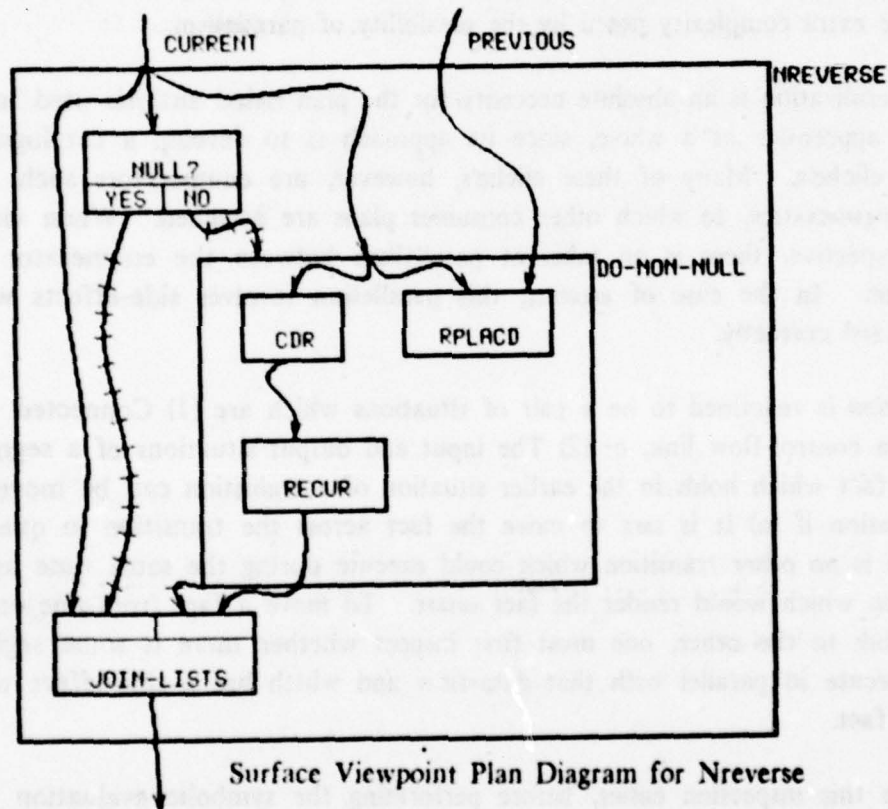
The source of the problem is that REASON does not know whether INDEX-1 and INDEX-B are equal or not since INDEX-B is an anonymous object. Therefore, REASON engages in a case analysis, splitting into two cases: (1) INDEX-1 equal to INDEX-B and (2) INDEX-1 distinct from INDEX-B. Each of these case gives the desired results rather directly. If the two indices are distinct then the BUCKET assertion above is SAFE. Similarly, all the other supporting assertions are SAFE, leading to the result that the membership assertion itself is SAFE; i.e. in this case, the membership of ENTRY-B in the table is unaffected by the changing of the bucket since it is in a different bucket.

In the other case, INDEX-1 is equal to INDEX-B and thus BUCKET-1 is identical to BUCKET-B. An *identification* (see Chapter 4) of BUCKET-B to BUCKET-1 is performed, leading to the conclusion that ENTRY-B is a member of BUCKET-1 in S-IN and, therefore, by the specs of LIST-DELETE, ENTRY-B is also a member of BUCKET-2 which is then stored into the TABLE. As above, this leads to the conclusion that ENTRY-B is a member of the table in the output situation.

Section 11.7: Pseudo Parallelism

As I have mentioned plan diagrams allow a weak form of parallelism. Although I am not at this time interested in the extra problems (and opportunities) presented by parallel execution, I have found this parallelism a convenient way of capturing some generalities of sequential processes. For example, in representing the most general form of the `BINARY-TREE-TRAVERSAL` plan fragment, we found parallelism allowed us to represent the many possible traversal orderings in a single plan diagram.

However, parallelism presents special difficulties when side-effects are introduced into the programming discipline. Consider the plan diagram for MacLisp's `NREVERSE`:



Without a control-flow link ordering the execution of the `RPLACD` and the `CDR` segments, there is no guarantee that one segment will execute before the other. Indeed, there is no information at all in this diagram about the mutual ordering of these two segments. Thus, it is necessary to regard them as executing in parallel and, therefore, capable of destructive interference.

For Complex Program Understanding

The plan diagram formalism regards any data-flow as taking a finite amount of time. In fact, since data-flows might be implemented by a pathway of many segments as in a `QUEUE-AND-PROCESS` plan, the time involved might be considerable. Therefore, it is also possible for the `RPLACD` to have a destructive interference with the data-flow to `NREVERSE`.

It follows that the transition analysis which I have discussed so far is too simple, since it has been conducted under the unstated assumption that plan diagrams are interpreted in a strictly sequential manner. Under this assumption all data flows preserve all properties and the only transition analysis required is at the transitions representing segments with side-effects. This will now have to be generalized to take account of the extra complexity posed by the possibility of parallelism.

This generalization is an absolute necessity for the plan based analysis used in the programmer's apprentice as a whole, since its approach is to develop a catalogue of programming cliché's. Many of these cliché's, however, are enumerators such as a `TRAILING-POINTER-ENUMERATION`, to which other consumer plans are attached. When viewed from this perspective, there is an inherent parallelism between the enumerator and consumer plans. In the case of `NREVERSE`, this parallelism involves side-effects which must be analyzed correctly.

A *transition* is redefined to be a pair of situations which are (1) Connected by a data-flow or a control-flow link, or (2) The input and output situations of a segment. In general, a fact which holds in the earlier situation of a transition can be moved to the later situation if (a) It is `SAFE` to move the fact across the transition in question and (b) There is no other transition which could execute during the same time as the one in question which would render the fact `UNSAFE`. To move a fact from one end of a data-flow link to the other, one must first inspect whether there is some segment which can execute in parallel with that data-flow and which has a side-effect which threatens the fact.

To make this inspection easier, before performing the symbolic evaluation of a plan diagram, `REASON` first analyzes the data- and control-flows, breaking the diagram up into separate *paths*. These paths can then be separated into sets of parallel paths. Two transitions one on each of two parallel paths can execute in parallel. Once this analysis of the plan diagram into parallel paths is completed, the transition analysis above can be generalized quite simply. Side-effect rules are now triggered by the combination of three types of facts (1) The existence of a side-effect,

(2) The existence of a fact in the earlier situation of a transition and (3) The possibility that the transition corresponding to the side-effect is on a path parallel to the one on which the transition occurs:

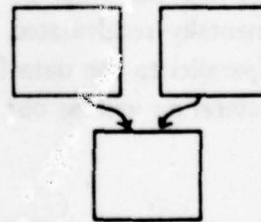
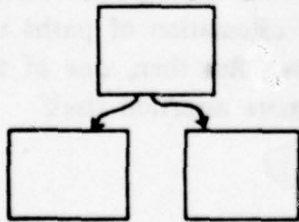
```
(Rule ((:f1 (:fact :s-1))
      (:f2 (Transition :s-1 :s-2))
      (:f3 (Onpath :path-1 :s-1 :s-2))
      (:f4 (side-effect :obj :s-3 :s-4 :new-fact))
      (:f5 (Onpath :path-2 :s-3 :s-4))
      (:f6 (Parallel :path-1 :path-2))))
```

appropriate transition processing

)

The actual analysis of the plan diagram into paths is rather simple. It begins by identifying *path joining segments* and *path splitting segments*, i.e. those segments at which two flows (data or control) come together at a single segment and those at which two flows diverge from a single segment. Segment execution can only begin when all the inputs are present; thus, when two data flows join at a segment a synchronization point is established. Similarly, since no output leaves a segment until all the outputs are ready, a synchronization point is established at segment output as well.

Path Splitting Segment



Path Joining Segment

Path Joining and Splitting

For Complex Program Understanding

When a path splitting segment is noticed, two new path names are created, one for each diverging flow. The splitting segment is declared to be the head of both of these paths and the paths are declared to be parallel. Similarly, when a path joining segment is noticed it is declared to be the tail of both paths entering it, and these paths are declared parallel. A segment which is entered by only a single flow, or by several flows each of which originates at the same segment is declared to be on the same path as the segment from which the flows came. This last step, however, is made to depend on the absence of other entering flows, so that if new flows are added to the diagram, new paths will be recalculated.

Several other rules are also involved in the calculation. For example: Two segments which are at the terminal end of conditional-control-flow links originating from the same segment are on separate but non parallel paths. A pair of paths is parallel if it consists of one path internal to each of two segments where the two enclosing segments are on parallel paths.

Consider `NREVERSE` again; a bug exists if there is no control-flow ordering the execution of the `RPLACD` and `CDR` segments. In the path analysis, these two segments will be analyzed to be on parallel paths; this will lead to the conclusion that there might be destructive interference between the `RPLACD` and the data flow to the `CDR`. The side-effect rules conclude that it is possible that the flow does not preserve the `REST` property.

However, this depends on the assumption that there are no further flow links ordering the two segments. If the programmer should intervene, adding a control-flow link to make the `RPLACD` follow the execution of the `CDR`, then this assumption will be violated and the paths incrementally recalculated. In the new calculation of paths the `RPLACD` will not be on a path parallel to the data-flow to the `CDR`. But then, one of the facts supporting the `UNSAFE` declaration will be *out*, owing the `UNSAFE` assertion itself.

Chapter 12: Reducing Complexity in Side Effect Analysis

In describing data-structures I defined a notion of structuresharing in a recursive-structure. Much of the complexity in reasoning about side effects occurs in recursively defined structures which share some substructure. Suppose that we know of the existence of two lists, and one of these is side-effected; given what we have developed so far, we must consider the possibility that this side-effect will change some properties of the second list as well. However, if we knew that the two structures were disjoint, then this possibility would be eliminated, reducing the complexity considerably. [Burstall, 1972] introduces some techniques for reasoning about side-effects which use this notion of disjointness to advantage. I will extend that notion in this chapter.

Let us examine in a bit more detail why this is true. The following is a side effect rule for list membership:

```
(Rule ((:f1 (side-effect :list-1 :s-in :s-out (first :list-1 :obj-1)))
      (:f2 ((Member :list-2 :obj-2) :s-in)))
      (If-possible (:f3 ((sublist :list-2 :list-1) :s-in))
        then (assert (not (safe-from :f1 :f2)) (list-mem :f1 :f2 :f3))
        else (assert (safe-from :f1 :f2) (list-mem :f1 :f2 :f3))))
```

this rule is derived from the following definition:

```
(Relation (Member List Object)
  (definition: (Member List Obj)
    (<=> (Or (First List Obj)
      (There-is (:sub) (Sublist List :Sub)
        such-that (Member :Sub Obj))))))
```

Suppose that we know that LIST-1 and LIST-2 do not share any structure; we can determine quite simply that it is not possible for the side-effected list to be a sublist of the other. This removes the need to conduct a thorough investigation as outlined in the preceding sections.

I will present in this section a hierarchy of classification for side-effects, properties and the degree of sharing exhibited by recursive-structures. Given that we have seen how lists and trees can be defined as special kinds of recursive-structures, this classification will be applicable to most of the useful structures of LISP programming. The purpose of this classification is to use the level of sharing to limit the degree to which side-effects to one structure can effect properties of the other. Similarly, the classification of properties into levels isolates properties of a higher level from less powerful side-effects.

Actually sharing is not as important as the lack of it: disjointness. I have identified three types of disjointness which have some utility. I have previously defined *structure sharing* as having a node in common. *Structurally disjoint* structures are those which do not share structure. For lists this means that no sublist (the transitive closure of *cdr*) of the two lists is shared.

Often we will have non-recursive structures such as hash-tables whose parts are recursive structures. For such objects, we define structure sharing in the obvious way: namely two objects share structure if there is a part of the first and a part of the second which share structure. Thus, a hash-table and a list share structure if one of the table's buckets shares structure with the list. They are structurally disjoint if there is no bucket of the table which shares structure with the list.

The next type of sharing is termed *value sharing*. Recall that the nodes of a recursive-structure can have other parts (called *values*) besides those which represent the immediate-children of the node. A list is a recursive-structure which has a *value* at each node called the *first*. Similarly some types of binary-trees have a value at each node. (See Chapter 10 for a review of these notions). When there is an object which is a value of two recursive structures, we say that there is *value sharing*; conversely, if there is no such object we say that the structures are *value disjoint*. Notice that if two objects share structure, they then must share values; since they have at least one node in common, the value of this node is a value of the two structures; therefore they *share values*. It follows that if two structures are *value disjoint*, they are also *structurally disjoint*.

Two objects are *totally disjoint* if (1) The objects are both structurally and value disjoint and (2) All objects pointed to by each node are totally disjoint. For example, two lists are totally disjoint if they share no sublists, if the members of the lists are distinct, and if as well the members of the first list are totally disjoint from the members of the second.

It follows that if two recursive-structures are totally disjoint, then side-effects to the one can not effect the other. Unfortunately, although total disjointness is not completely rare, it is not the most common event either. In particular, lists frequently have common members such as atoms, and are, therefore, not totally disjoint. However, structure sharing is also reasonably rare. The sharing of list structure presents enormous opportunities for powerful interactions and thus for bugs; therefore, most programmers avoid sharing except in those cases where the power is actually desired. Most side-effects have very limited scope as long as there is no structure sharing.

To begin, let us classify side-effects in a manner similar to that used for structure-sharing. We call side-effects *strictly structural* if they only affect the immediate-children property of some node of the structure. `RPLACD` is the simplest such side-effect, although `LIST-INSERT` and `LIST-DELETE`, `NREVERSE`, `SORT`, `NCONC` and various other built-in functions of MacLisp also are strictly-structural side-effects. Notice that a strictly structural side-effect to one structure will not affect any property of another object which is structurally disjoint from the first.

We may also identify *strictly value* side-effects such as `RPLACA` which only change value parts of a recursive-structure. If two structures are structurally disjoint, then a strictly value side-effect to one will not affect properties of the other. A *structural side effect* is one which consists only of strictly structural and strictly value side-effects. A sort program which works by changing both `CAR` and `CDR` pointers in a list is an example of a structural side-effect. Again if two objects are structurally disjoint, a structural side-effect to one will not change any property of the second.

Finally, an *indirect side effect* is one which only changes properties of values of a recursive-structure. For example, a marking graph traversal procedure which sets the `MARK` property of the value of each node is such an indirect side-effect procedure. If two objects are value disjoint, then indirect side-effects to one will leave properties of the other unchanged.

We say a structure is *isolated* if it shares structure with no other objects. We may divide this into the types used above, referring to structural, value, and total isolation. Most routines which build new structures such as APPEND, or LIST create structurally isolated objects. This is important, since a structurally isolated object is relatively safe to side-effect; often programs will create a copy of an object and side-effect the copy as a means of guaranteeing that unwanted interactions do not result.

Finally we come to a classification of properties. We can notice that some properties such as SUBLIST or LENGTH only depend on the recursive structure of the object, and not on the VALUES at each node. We call these *strictly structural properties*. More commonly, properties such as MEMQ, depend both on the recursive structure and on the identity of the various VALUES present at each node, but not on any property or sub-structure of these values; these are called *value dependent properties*. Finally, there are properties which depend both on the structure and on mutable properties of the objects present at each node. MEMBER (as opposed to MEMQ), for example, depends on the structure of the list as well as on the structure of the objects pointed to by the list; if the list (A B C) is a MEMBER of the list L-1 then the LISP invocation:

```
(MEMBER '(A B C) L-1)
```

will return a NON-NIL answer, namely the list (A B C), which is a member of L-1. If this list is then RPLACD so that its first element is X, then (MEMBER '(A B C) L-1) will return NIL. Let us call such a property a *value indirect property*.

A side-effect at one level can not effect a property of a lower level. For example, a value side effect cannot affect a strictly structural property. An indirect side-effect cannot affect a structural property.

These observations can be summarized by several simple rules of the following form:

```

(rule ((:f1 (Side-effect :obj-1 :s-in :s-out :se))
      (:f2 (Structural-Side-Effect :se))
      (:f3 (Structurally-disjoint :obj-1 :obj-2))
      (:f4 ((:fact :obj-2) :s-in)))
  (Assert (Safe-from !:f1 !:f4)
    (Structure-disjoint :f1 :f2 :f3 :f4)))

(rule ((:f1 (Side-effect :obj-1 :s-in :s-out :se))
      (:f2 (Structural-Side-Effect :se))
      (:f3 (Structurally-isolated :obj-2))
      (:f4 ((:fact :obj-2) :s-in)))
  (Assert (Safe-from !:f1 !:f4)
    (Structure-isolation :f1 :f2 :f3 :f4)))

```

Notice that since these rules assert that a particular property is safe, they remove the need to engage in the more complicated analysis shown in the last chapter. A large percentage of side effects have very limited range of effect precisely because there is a strong limit on structure sharing. Most of the time one of the above rules will fire and REASON's work will be done. In some rare cases, the more complex and thorough analysis will be required.

This approach requires a classification of side effects into the various levels and a similar classification of properties. Relation-definitions provide the basis for these classifications. To decide whether a property is structural one need only determine whether it depends on the node property of the object, given that the object can be viewed as a recursive-structure. Similarly, if the property depends on any value pointer of a node it is a value property. If it depends on properties of the value objects it is an indirect property. For example, consider the MEMBER relation for LISTS:

```

(Member List Object) <=> (Or (first List Object)
  (Member [rest list] Object))

```

Since the property depends on FIRST which is the value pointer for lists, the property is a value property. Also since it involves a recursive definition involving the rest (which is the immediate-child pointer for lists) it is a structural property. IMAGE-IN in a ALIST is a indirect property as shown by its definition:

240 Reducing Complexity in Side Effect Analysis

```
(Image-in Alist Key Value)  
<*) (There-is (:el) (Member Alist :el)  
      such-that (And (Key :el Key)(Value :el Value)))
```

Since this definition depends on `MEMBER` which is both a structural and value property, `IMAGE-IN` is also a structural and value property. However, in addition it depends on the `KEY` and `VALUE` parts of `MEMBERS`, which are values; therefore, it is an indirect property. Side-effects may be categorized using a similar analysis of defined relations.

Chapter 13: Reasoning About Program Modifications

In the previous chapters I have shown how REASON analyzes a program, maintaining an explicit representation of all logical dependencies. Although such an explicit representation is costly in terms of space consumption, I will show in this chapter how that cost is repaid during the process of program modification. It cannot be overemphasized that this concern has been the driving force behind my design decisions. The price of software maintenance is the most rapidly escalating part of computer costs and the one with the least likelihood of decreasing. An expensive tool which effects a ten percent reduction in software costs would repay its cost quite many fold.

When REASON has analyzed a program it has a very rich knowledge structure annotating the program text. This structure includes a complete record of the proof of all pre-requisite and achieve goals, purpose links summarizing the inter-relationship between the spec clauses of the sub-segments and the main segment, links to implementation methods, defined relations and other knowledge about the data-objects of the program, and finally a recognition map connecting fragments of the program to the standard plans of the library. Such standard plans, in turn, are organized into specialization hierarchies in which, for example, LIST-ENUMERATION is regarded as a specialization of the ENUMERATION plan for general recursive-structures.

When a program modification is proposed, REASON uses this rich knowledge structure to discover how far the effect of the proposed modification will propagate. Typically, there is some decomposition of the program in which the change has effect only within a particular segment's boundaries, leaving unchanged most of the logical structure outside. For example, if the method of representing a SET is changed from LISTS to ARRAYS, then only the ENUMERATION part of the program will be modified. By looking at the temporal viewpoint of the program we can regard the ENUMERATION as a separate segment which produces a temporal-collection of the MEMBERS of the SET. This is true in both the new and the old versions of the program. Thus, we know that the rest of the program (the part outside the ENUMERATION) is not affected.

The goal in analyzing a program modification is to be able to use the logical analysis of the old program to help understand the new program. As the programmer edits the old version, REASON attempts to follow the chains of dependencies to see what requirements of the old structure are no longer met. If there are no such broken chains, then the modification is merely an addition of some new behavior

For Complex Program Understanding

which can be analyzed by the mechanisms of the previous chapters. We will now look at how REASON determines what is affected.

Most changes are relatively straightforward to analyze. For example, consider what must be done if a new expect clause is added to a segment. First, the dependencies linking the expect clauses to the segment's *applicable* assertion must be rebuilt to include the new expect. Then a proof of the new expect clause must be undertaken. If this succeeds, the segment will be declared applicable and no interaction with the user is required. If not, the *APPLICABLE* assertion for the segment will be *out*.

The Truth Maintenance System can be requested to signal every time a particular fact changes status from *in* to *out* or vice versa. As REASON evaluates a plan diagram it makes such requests for every expect clause of a sub-segment and every assert clause of the main segment. Also such requests are made for the *APPLICABLE* assertions for each sub-segment and for the main segment. Thus, when analyzing the effect of adding a new expect, REASON is first signalled that the sub-segment is no longer applicable. If the proof of the new expect clause succeeds, TMS signals that the status of the applicable assertion is now *in*. Thus, REASON knows that everything is alright. If not, REASON reports that the segment is no longer applicable. However, all segments which depend on the modified segment will also become inapplicable; REASON collects these signals as well. What to do with this information is the province of discourse expertise not yet present in the apprentice system.

Removing an *assert* from a sub-segment presents a similar problem, although there is a new opportunity. Since REASON has already built purpose links, it is a simple matter for it to consult these before doing anything else. The purpose links tell REASON that the *ASSERT* provides support for various sub-segment *EXPECT* clauses and main segment *ASSERT* clauses. Each of these is examined to see if they have other support which is independent of the clause being removed. If each such clause has independent support then the change has no effect; REASON tells the user that everything is in order. Otherwise, it issues a warning, saying which dependent segments are affected. If more information is desired, REASON follows through the actual justifications to build a trace of the broken proof.

In the case where one clause is deleted and another is asserted, REASON waits until both actions have been performed before checking to see which APPLICABLE assertions have changed status. Otherwise it handles matters as above.

When a data-flow link is changed, the assignment of objects to the segment's input ports must be updated; those expect clauses which mentioned the affected port must be recreated with the correct objects substituted in. The justifications linking expect clauses to the segment's APPLICABLE assertion must then be rebuilt. REASON then proceeds as in the case where an expect clause is changed. Notice that in all these cases when a part of the plan diagram is removed, the TMS *outs* all goals and conclusions which followed from the *outed* statement.

When a relation of an object-type is redefined, similar effects take place. The rules corresponding to the relation definition not only make a deduction, but as with everything else in REASON, they provide a justification for the deduction. In the case of defined relations, the justification points to the assertion in which the relation-definition is stated. Thus, if the relation is changed, the old definition goes *out* and facts following from the definition also lose support.

More commonly, however, the programmer will not change a relation-definition, but will rather create a new object-type in which a different definition appears. If an object of the old object-type was deduced to have a particular property, then the justification for this will point to the most specific object-type in which the relation is defined. For example, the membership relation for ALISTS is defined at the level of LISTS; the justification, therefore, points to the assertion stating that the object's type is LIST and not to the assertion stating that it is a ALIST. In contrast, the IMAGE relation (x is the IMAGE of y in the ALIST $L-1$ if there is a pair whose left is x and whose right is y and that pair is a member of $L-1$) is defined at the level of ALIST; any deduction arising from the IMAGE relation would depend on the assertion stating that the object is an ALIST. Thus, if the programmer changes the type of an object, all deductions which depend on its having an object-type which it no longer holds will lose their support and go *out*. However, rules corresponding to the definitions of new relations corresponding to the new type might trigger, bringing in new facts. Finally, if the object is changed from one sub-type to another, it is possible that no important relation is defined at the more specific level and, therefore, nothing significant will happen.

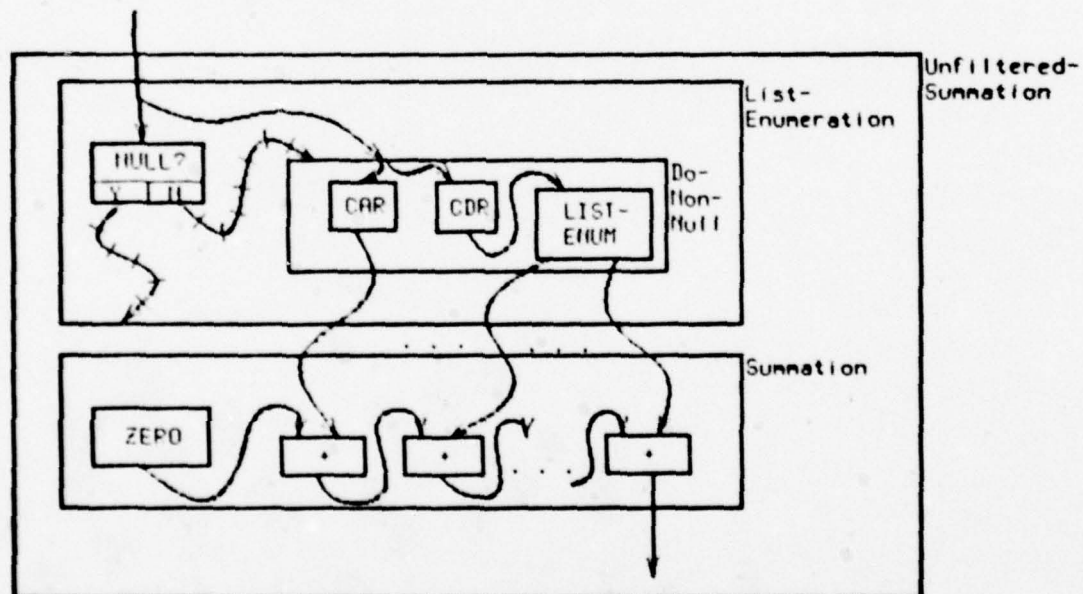
In general, then, the pattern of reasoning at this level is quite clear. A plan editing program is instructed by the user to add, delete, or change some feature of a plan diagram or some part of an object description. This action may cause some facts to change status from *in* to *out* or vice versa. If an APPLICABLE assertion for either a sub-segment or the main segment winds up being out after all effects have been propagated by the TMS, then REASON warns the programmer that an error has been introduced.

Section 13.1: Updating The Recognition Map

When the apprentice first analyzes a program, it builds a recognition map explaining how the program uses standard library plan fragments to achieve its goals. A program modification will typically force the system to rebuild this map to reflect the new situation. Fortunately, simple modifications to program structure do not affect the recognition map in a drastic manner. For example, consider the following program:

```
(defun Accumulate-Salary (List-of-Employees)
  (do ((l List-of-Employees (cdr l))
      (sum 0))
      ((null l) sum)
      (setq sum (+ sum (car l)))))
```

This is a simple, unfiltered summation program. The LIST is a list of RECORDS, where each RECORD has the SALARY in the FIRST position of the record. This is diagrammed as follows:



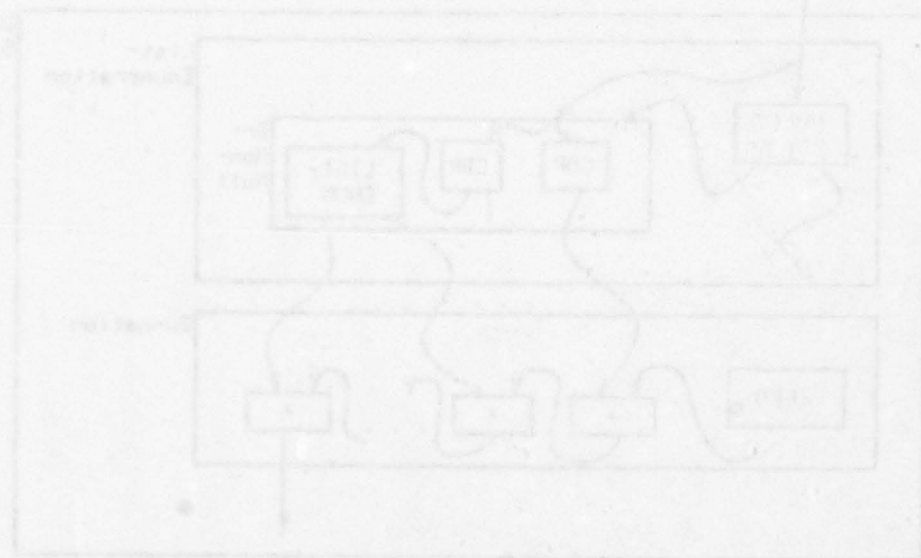
Unfiltered Summation Program

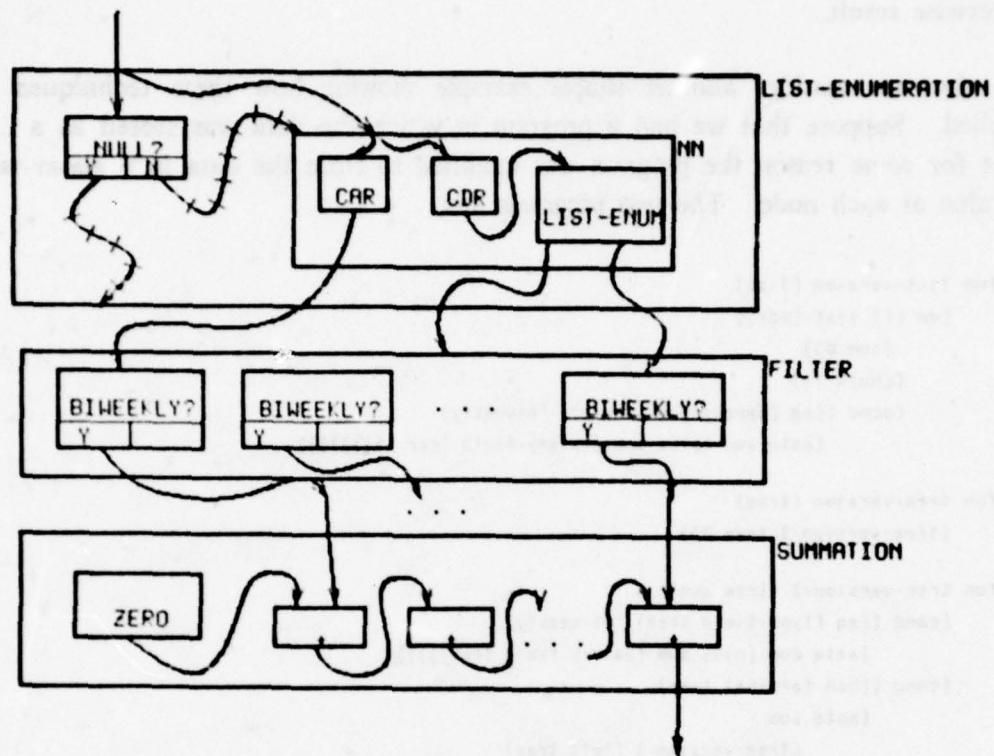
For Complex Program Understanding

Let us consider various simple modifications which might be made to this program. Suppose, for example, that we put in a test so that we summed only the bi-weekly employees:

```
(defun Accumulate-Salary (List-of-Employees)
  (do ((l List-of-Employees (cdr l))
      (sum 0))
      ((null l) sum)
      (cond ((eq (caddr 1) 'bi-weekly)
             (setq sum (+ sum (caddr 1)))))))
```

where the second field of each record is the employee-type. Of course, the effect of this change is to filter the inputs to the SUMMATION segment. The recognition proposer would suggest that the recognition of the SUMMATION segment is still correct and that the recognition of the LIST-ENUMERATOR is still correct. (The recognition proposer [Rich, 1977] is outside the scope of this thesis). However, the temporal-collection input to the SUMMATION segment is now changed; the plan recognizer suggests that the new segment is a FILTER segment interposed between the ENUMERATOR and the SUMMATION routine.





Filtered Summation Plan

Thus, the recognition of two parts of the plan remains the same; only the **FILTER** section and the temporal-collection data-flows change at all. We can easily change our understanding of the overall effect of the plan to reflect the addition of the **FILTER** simply by re-evaluating what collection of objects flow into the **SUMMATION** segment in the new plan.

The basis for this separation is Water's [Waters, 77] observation that recursive programs (including loops) can be broken up into a temporal decomposition by inspection of the pattern of data- and control-flow links. Thus, when a recursive program is modified, the system checks to see whether the clues used previously in forming the segmentation are still *in*. If so, it only tries to form segments for the new code. Although, my system and Waters' are not yet interfaced into a unified apprentice system, the discipline of explicit recording of all important control information can serve to make the interface a matter of less complexity than would

For Complex Program Understanding

otherwise result.

Let us consider another simple example showing how these techniques can be applied. Suppose that we had a program in which the data was stored as a LIST and that for some reason the program was modified to store the data in a BINARY-TREE with a value at each node. The two programs are:

```
(defun list-version (list)
  (do ((l list (cdr))
      (sum 0))
      ((null l))
      (cond ((eq (type-field (car l)) 'biweekly)
             (setq sum (plus sum (salary-field (car l)))))))

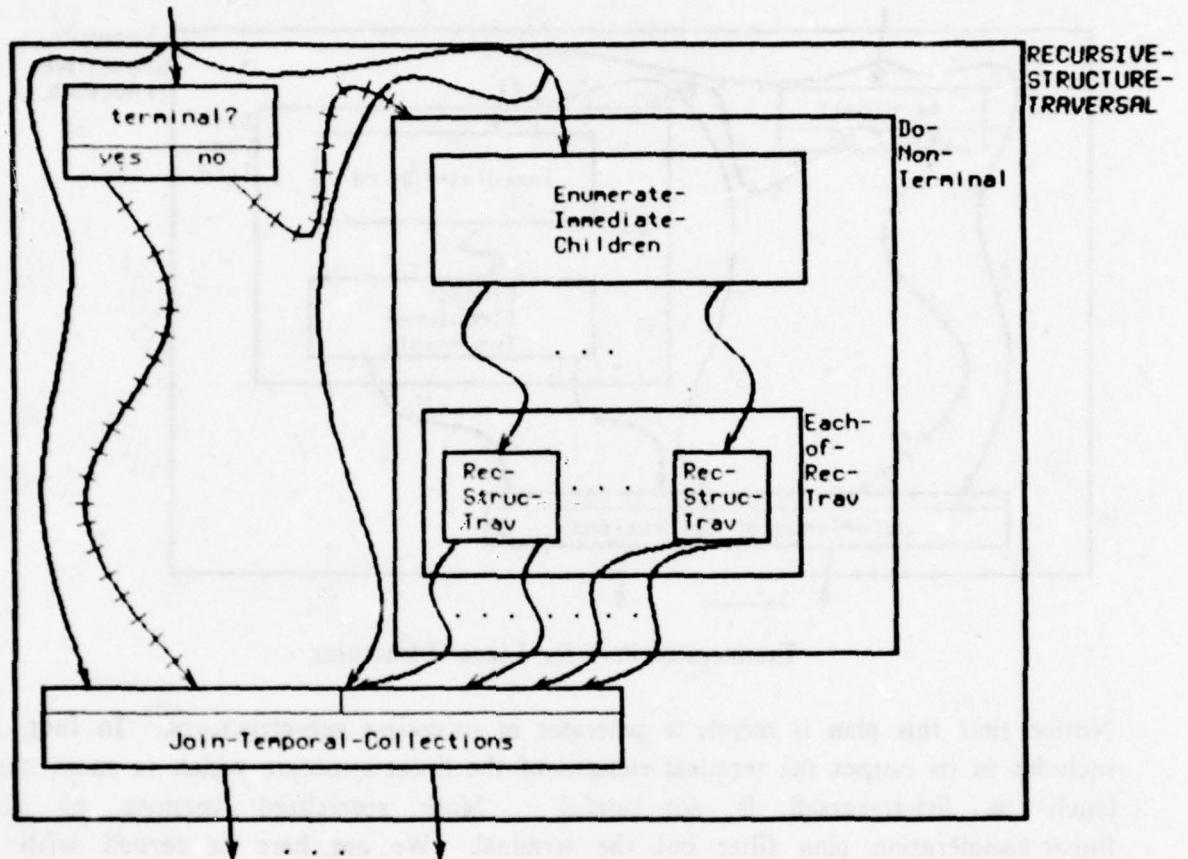
(defun tree-version (tree)
  (tree-version-1 tree 0))

(defun tree-version-1 (tree sum)
  (cond ((eq (type-field tree) 'bi-weekly)
         (setq sum (plus sum (salary-field tree))))
        ((non-terminal tree)
         (setq sum
                (tree-version-1 (left tree)
                                (tree-version-1 (right tree) sum))))
        (t sum))
  sum)
```

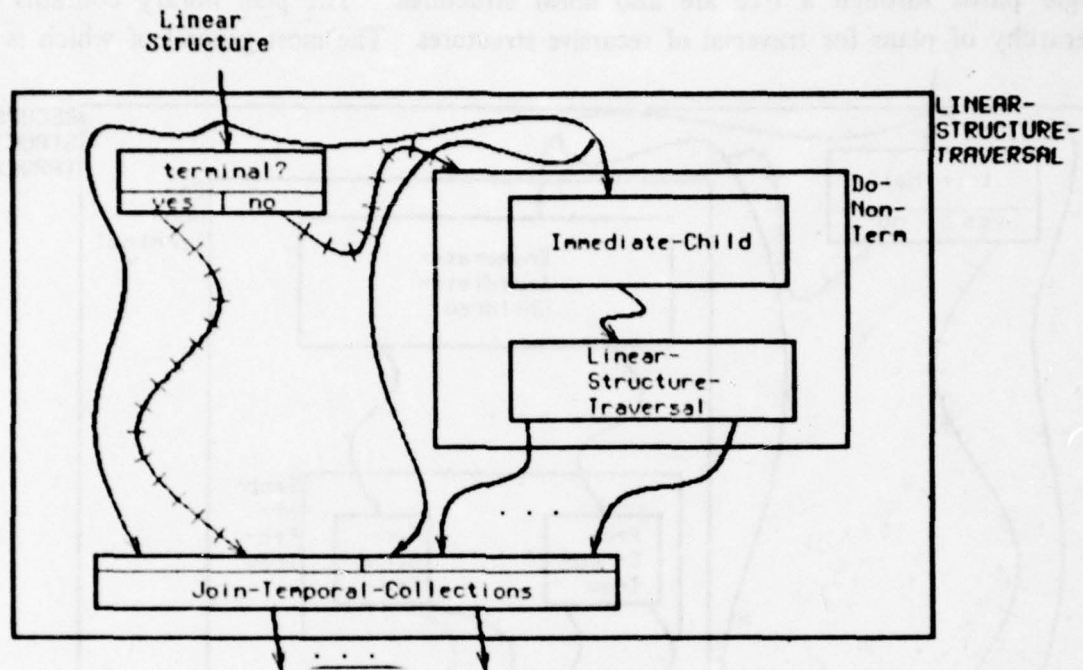
The second program can be analyzed as a COMPOSITION of a TREE-TRAVERSAL, a FILTER, and a SEQUENTIAL-SUMMATION plan. Similarly, the first program can be separated into a LIST-ENUMERATION and the same FILTER and SEQUENTIAL-SUMMATION plan. In making this coding change many surface details change, however, much of the deep structure of the program remains constant. As the apprentice analyzes the modification, it will have to rebuild the recognition mapping since some of the details of the recognition have changed. However, those details which don't change are represented as facts in the data base which stay *in* throughout the whole process. Thus, those deductions which are based on facts of the analysis which are not changed between the two versions stay *in* themselves and do not require any further deductive effort.

Let us look again at an example shown at the beginning of this thesis in which a HASH-TABLE is changed from a LINKED-LIST representation for the BUCKETS to a REMASHING scheme in which the cells of the array form the data structure to be searched. We can easily see that both the LIST and the SET-OF-CELLS form an ACYCLIC RECURSIVE STRUCTURE of NODE-DEGREE 1. We call such structures *linear* structures and observe that arrays and

single paths through a tree are also linear structures. The plan library contains a hierarchy of plans for traversal of recursive-structures. The most general of which is:

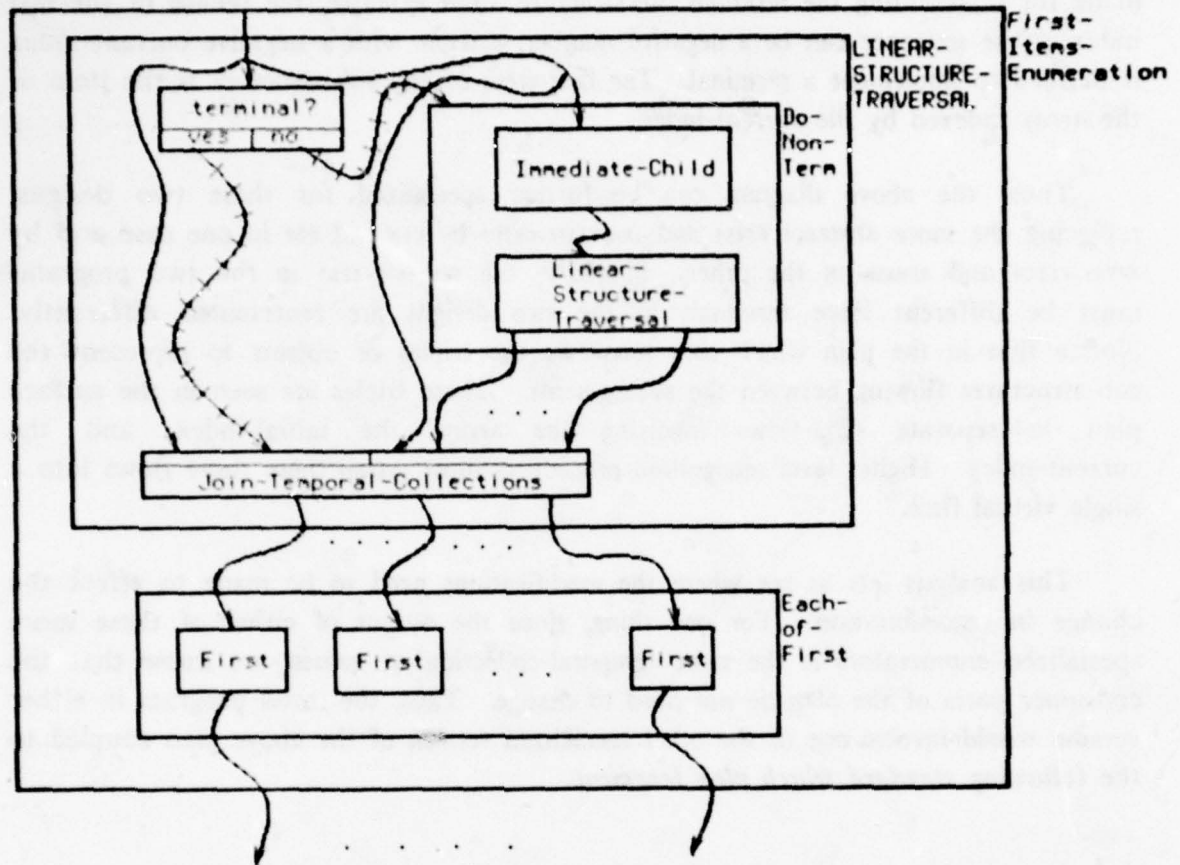


For linear-structures there is another more specialized plan which replaces the plan for enumerating the immediate-children by the more simple plan IMMEDIATE CHILD which fetches the unique immediate child of the current node. Thus, we get:



Enumerator Plan for Linear Structures

Notice that this plan is merely a generator of successive sub-structures. In fact, it includes in its output the terminal element of the linear-structure which in most case (such as list-traversal) is not useful. More specialized versions of the linear-enumeration plan filter out the terminal. We are here concerned with a particular-type of linear-structure, those with a value set of size 1 (i.e. we want there to be a unique first value of each sub-structure). Most often we want to augment the specialized linear-enumerator with an operation to fetch the first value of the sub-structure.



Enumerator of Values of A Linear Structure

This plan can be specialized in many ways depending on the nature of the linear-structure. However, the specialization is always concerned with the representation of the linear-structure; i.e. with the particular means of fetching the immediate-child and the particular means of fetching the first item of the sub-structure enumerated. In the two cases we are considering these details are quite different. In the case of LISP lists, the immediate-child operator is *cdr* and the first operator is *car*.

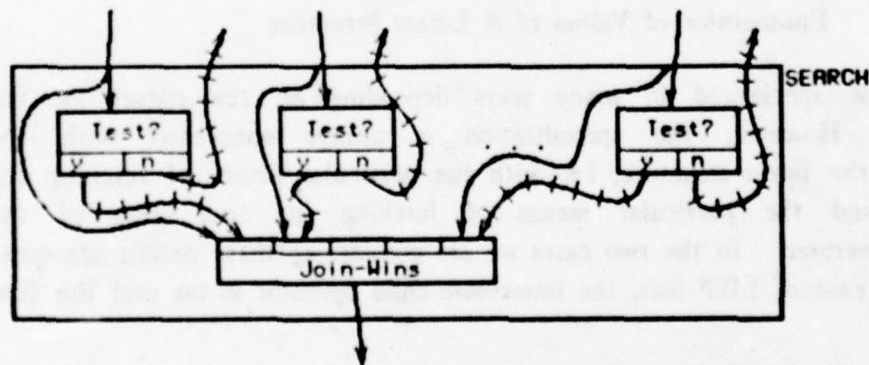
However, in the case of the *REHASHING* scheme, things are a bit more complicated. The sub-structures being enumerated are represented by a triple consisting of an array, an *INITIAL-INDEX* and a *CURRENT-INDEX*. The next sub-structure is the object implemented by the triple consisting of the same array and initial-index but with a new current-index which is the rehash of the old current-index. Special provision is

For Complex Program Understanding

made for representing the terminal sub-structure. For example, the rehash of the last index in the sequence can be a negative number; a triple with a negative current-index is defined to implement a terminal. The first item of any sub-structure is the item of the array indexed by the current-index.

Thus, the above diagram can be further specialized for these two designs, replacing the more abstract FIRST and IMMEDIATE-CHILD by CAR and CDR in one case and by ARRAY-FETCH and REHASH in the other. Similarly, the TERMINAL-TEST in the two programs must be different since terminals in the two designs are represented differently. Notice that in the plan which uses REHASH we use triples of objects to represent the sub-structures flowing between the subsegments. These triples are seen in the surface plan as separate data-flows involving the array, the initial-index, and the current-index. Higher level recognition procedures must group these three flows into a single virtual flow.

This analysis lets us see where the modifications need to be made to effect the change in representation. For one thing, since the output of either of these more specialized enumerators is the same temporal-collection of values, we know that the consumer parts of the plan do not need to change. Thus, the LOOKUP program in either version would involve one or the other specialized version of the above plan coupled to the following *standard search plan fragment*.



The Standard Search Plan

The key observation, however, is that the SEARCH plan can be seen to have no dependencies linking it to the design choice of the data-structure traversed by the ENUMERATOR. Indeed, only the TERMINAL-TEST, FIRST and IMMEDIATE-CHILD sub-segments of the ENUMERATION plan depend on the design choice. When the programmer proposes to change the representation of the bucket from lists to a different type of linear-structure, the apprentice, can immediately determine the extent of the effects. The entire SEARCH part of the plan is safe and the only part of the ENUMERATION which is affected is the FIRST and REST parts which in the current version are the CAR and CDR segments, and the TERMINAL-TEST which in the current version is the test NULL?. Based on these observations, the apprentice can describe in high-level terms what segments need changing.

In a more advanced version of the apprentice I expect there to be at least enough synthesis expertise to chose the correct specialization of LINEAR-ENUMERATION for traversing the REHASHED-CELLS data-structure. The apprentice could use this newly selected plan to inform the programmer how to change his program to conform to the new design. In any event, the ability to decompose the program into both temporal and surface viewpoints allows the apprentice to treat the above modifications as incremental just as would any reasonably skilled programmer.

Chapter 14: Conclusions

Section 14.1: Good Decisions

REASON's design deviates from that of standard verification systems such as [Igarashi, et. al, 1973] in many ways. REASON is intended to be a part of an interactive programmer's apprentice system. It must function in a number of different contexts including interactive design, plan modification, and verification and it must service the needs of different communities of programmers using a variety of languages. These requirements led to several novel design decisions.

In building the apprentice, it seemed essential that the system not be primarily concerned with the actual program text. The primitives of a programming language are too low level to worry about during the early (and probably later) stages of design. Rich and I concluded that the system's formalism should be quite simple, consisting of program segments connected by control and data flow; these seemed to be the abstract notions which programming language primitives are intended to achieve. Also we thought that this formalism would be a convenient one in which to capture the teleological notions which constitute a *plan*.

The plan diagram formalism has, so far, done what it was intended to do. However there are some concepts which it can't handle. The formalism has no place for a procedure which, like an interpreter, manipulates the representation of another procedure, converting list structure into new data and control flow links. The notions of data and control pathways takes us almost to this goal, but more work needs to be done. Similarly, we currently have no way of describing interrupts or synchronization primitives. However, these were beyond the scope of our original goals; indeed, the kinds of programs which we wished to attack are simply and conveniently described by plan diagrams.

A second novel design choice was to use a symbolic interpreter rather than a verification condition generator. The apprentice is intended to help a programmer design systems; this is a somewhat chaotic process in which a programmer might change his design frequently, moving segments, changing data flows, and adding in new segments to perform tasks which had been overlooked. To do this effectively, the programmer will need to have "snapshots" of the computation, so that he can ask whether a property holds at a particular point of the computation. REASON's use of situations satisfies this need while adding yet another advantage. The deductive

apparatus in REASON is completely integrated with the symbolic interpreter, allowing simplification and partial deductions to be made as the interpretation of the plan proceeds.

A third unusual feature of my approach is that many low level language features simply do not appear in REASON's plan diagram formalism. Assignment to variables is regarded as a means of implementing a data flow; *while* loops, etc. are all captured by recursion. I have grown to find this means of expressing a program quite natural and simple and think that a front end for the system could be engineered to make plan diagrams a very natural vehicle for communication between the apprentice and the programmer.

One future development would be to use a graphics system to allow the programmer and the apprentice to communicate pictorially; the system would generate the assertions of a plan diagram internally. The system would be able to display standard library plans, modifying and specifying them on command from the programmer. Systems could be designed by cutting and pasting pictorial plan diagrams on a tv screen. This would require the programmer to learn the vocabulary of the plan library, but I think this would be advantageous. The plan library gives names to the standard patterns of programming; if programmers began to think in terms of such notions, their task would be conceptually simpler. *Search, accumulation, tree traversal*, etc. are all more powerful conceptual terms than are *while, do*, etc.

Another unique feature of the programmer's apprentice project has been our emphasis on plan recognition and the development of a plan library. Much of this work is being done in a separate thesis by Rich [Rich, 1977,78], yet its influence on REASON has been considerable. My concern with the *temporal viewpoint* and the reasoning needed to support it is motivated by the needs of the plan library. This development has helped us develop a natural and powerful vocabulary for describing programs. Even if the rest of the project never reached fruition, the vocabulary itself is of great value and might form the basis for introductory programming classes. A student who learned to think in terms of standard plans, would probably have a much easier time mastering the basic skills of program design.

Turning now to my work on the reasoning system there are several decisions which I feel were positive. The use of Doyle's TMS [Doyle, 1978] as an integrating mechanism is well justified. The decision to place a great deal of emphasis on defined relations, allowing the user to state these declaratively is another idea which seems worth the extra effort of having the system translate these declarations into various kinds of procedures. The work I have begun on reducing the complexity of side effect analysis seems a quite promising outgrowth of the general approach of integrating the reasoning system with a knowledge base and an epistemology of programming concepts.

The main advance I feel that has been made in the current version of REASON is the task agenda protocol suggested by Doyle [Doyle, 1978b]. Although I have not yet fully developed the choice making protocols of the system, I believe that this offers the only route for building a system with evolving capabilities. Hopefully, future research will provide some insights into how to use this power to advantage. Finally, I think the initial work on modification reported here is moving in the correct direction. The use of temporal viewpoint plans to guide the system during modification sessions is a promising idea.

Although the current version of the system is still being implemented, I did succeed in getting the first version to do some fairly involved proofs. In the scenario of this thesis the programmer designs an associative retrieval system along the lines of Conniver's data base. The first version of REASON successfully completed proofs of all the routines used in this data base, including the fast intersect routine, the indexer, and the pattern matcher. In addition, it recorded the dependencies produced during these proofs and summarized them into purpose links. It was not overly fast, however, and it needed most of the 256K available on our PDP-10 to complete the longest of these proofs. Although this earlier system had some success, it was not the system I wanted. The newer system combined with the MIT LISP machine hardware will probably be a far more useful machine.

Section 14.2: Problems

I had hoped to implement more of the new system by now; yet each implementation stands on the shoulders of its predecessor. My experiences in implementing the earlier version of the system are worth mentioning. Originally I was primarily concerned with the efficiency of the system and, therefore, rejected the already existing general purpose problem solving language in favor of building mechanisms carefully tailored to the special needs of program analysis. As a result I spent a considerable amount of intellectual effort on issues at too low a level. As time passed, it became increasingly obvious that I was re-inventing the wheel.

One particularly painful aspect of this problem was the use of an unduly complicated context system (see [Rich & Shrobe, 1976]). REASON builds a situation tree representing the temporal behavior of the program; since it had to engage in hypothetical reasoning as well, I implemented an extension of the context mechanism of Conniver [McDermott, 1972] which allowed two dimensions of context. This was unfortunate; the mechanism was awkward and caused quite a few obscure bugs. Second, and far more significantly, the context mechanism is inappropriate for my purposes. However, since I was taking an incremental approach to the implementation, I didn't realize this until late in the first implementation at which point I was forced to stick with what I had.

There are two circumstance in which the context mechanism seems unusable. The first of these is plan modification; when a programmer modifies a plan by adding or deleting a segment or by changing a data or control flow link the succession of situations is changed. In such a circumstance, the context layers must be reorganized, often in ways which are precluded by artifacts of the context mechanism. Similarly, transition analysis is made cumbersome by the context mechanism which automatically moves facts forward. Some other process must intervene, erasing the fact in the context layer at which it ceases to be true. One can only begin to appreciate the hairy timing problems this can cause; to fully appreciate it, you should code up such a system and attempt to develop it. Finally, there was a representational problem; in order to give a justification for a fact in a particular context it was necessary to have a name for the context. To understand this name, the system had to have a map of which names come after which others. The justifications in the old system, therefore, were based on the situation tag representation while the reasoning system used contexts. The use of situation tags, TMS, and explicit control assertions is a far easier discipline to live with.

For Complex Program Understanding

The current system, however, has some irksome problems also. One of these is that it appears to be even more space consuming than the first implementation. Within the next few years the hardware revolution will make this concern irrelevant; in the mean time, however, experimentation is difficult. A more serious worry is that the current system's mechanisms for side effect analysis, although correct, are not as natural as I would like. The truth maintenance system ought to be able to use its justifications to determine which facts should move across a transition. McCallester and Doyle (private communication) have both suggested ideas for this kind of a process, but these have not been incorporated into the current design.

There are still important forms of reasoning which are outside of REASON's scope. Primary among these is reasoning about termination and the closely related concern of time complexity. Techniques described later in the literature review such as "ghost variables" might be quite easily integrated into the current system, but I have not yet examined this idea thoroughly. Reasoning about space consumption is another issue which I have not yet addressed at all. Finally, many of the powerful heuristic techniques for inductive proofs used in some other systems [Boyer & Moore, 1977] have not yet been integrated into REASON.

Section 14.3: Future Directions

I see this work growing in two directions at once. As I indicated in the introduction, my work can be viewed as a technical stepping stone for future work on self conscious systems such as that proposed in [Doyle, 1978b]. A next step in that direction is to build an interpreter for plan diagrams. This is, in principle, quite simple to do; it can follow the general pattern of the symbolic interpreter of this thesis. An interesting exploration would be to implement the symbolic interpreter as a plan diagram for the actual interpreter. Somewhat short of such fanciful exploration is to begin to develop proof strategies as plan diagrams which are executed by the interpreter. This will allow proof steps to be parts of more macroscopic actions within which they play well defined roles. These roles can then be categorized and used as the basis of both general and domain specific strategies. [Doyle, 1978b] discusses these ideas more fully.

REASON seems to have quite a bit of room for development within the apprentice system as well. First of all, there are numerous tasks described in this thesis such as modification, and recognition which are not yet integrated into the system. More interesting, however, are some avenues of exploration which we have not yet developed. One of these is the use of the reasoning system in more unconstrained recognition scenarios than I have presented here.

I would like the apprentice to analyze 4 or 5 pages of related LISP functions with almost no human intervention. Such a task would involve sophisticated problem solving strategies drawing on the powers of the reasoning system. In particular, it seems that this kind of recognition involves a certain amount of design expertise. One organization of such a system would have a heuristic recognition component use lower level clues to guess what high level design underlies the code. This design would then be elaborated by a program synthesis module (now being worked on by Rich [Rich, 1978]) working cooperatively with the reasoning system.

The next level task I would like to work on is the development of new expertise within the apprentice system. Currently, the system relies on a body of programming knowledge. As we now envision program synthesis, the apprentice can build a program when it knows plans appropriate for the synthesis task. This can go a long way if the knowledge base is extensive and sophisticated. However, introspection suggests that there is more to programming than just pasting together what one already knows. A direction of research which seems quite promising is to use the

For Complex Program Understanding

reasoning system to develop new plans through logical analysis, analogical reasoning, etc. Many of these ideas are being pursued by other researchers in other contexts. I will indicate these in the next section which reviews the related literature.

Chapter 15: A Survey of Related Work

[Waldinger & Levitt, 1973] point out that the first program verification methodology was worked out by Von Neumann [Von Neumann, 1963] and that validation is as old as software itself. In 1966, McCarthy and Painter [McCarthy & Painter, 1966] presented a proof of correctness for a simple expression compiler. Foundational techniques using the notion of a state vector (the vector of current values of all program variables) were presented in [McCarthy, 1962a,b, 63]. Indeed, the formal definition of Algol [McCarthy, 1964] was influenced by a concern for provability.

However, the modern interest in verification seems to date back to Floyd's pioneering work [Floyd, 1967] (independently [Naur, 1966] developed similar ideas). In this method, flowcharts are annotated by assertions which are believed to hold any time control passes through the annotated point of the program. An informal verification may be constructed by dividing the flow chart into control paths, showing that each assertion is a logical consequence of the earlier assertions and the intervening program steps on its path. The pairs of entrance and exit assertions state the I/O properties of the program. Normally, the programmer need only supply these assertions and one assertion, called the *invariant*, for each loop. This notion of correctness is called *partial correctness*, since it does not include a proof that the program terminates. The assertions are called *inductive assertions*.

Floyd also introduced a method for proving termination of programs. This proof is conducted separately from the proof of partial correctness and involves constructing a mapping between program variables and a well-founded set, i.e. a partially ordered set with no infinite descending chains. If a monotonically decreasing mapping can be constructed then the program must terminate. [Manna, 1969] formalized these results showing that the partial correctness of the program is equivalent to the satisfiability of a statement in first order logic and that total correctness is equivalent to the unsatisfiability of a second statement. Intuitively, the first statement says that there is a set of inductive assertions from which a partial correctness proof can be built; the second statement says that there is no set of assertions which would imply that the program halts with incorrect values.

C.A.R. Hoare [Hoare, 1969] extended Floyd's work by showing how it could be fit into a formal logical language. Hoare introduced the notation $P \{A\} Q$ to mean that if P is true before program A is executed, then Q will be true after A 's execution, (if A terminates). Hoare also presented several rules of inference for this system such as:

$$\frac{P \rightarrow R, R \{A\} S, S \rightarrow T}{P \{A\} T}$$

In later work, Hoare [Hoare & Wirth, 1973] presented an axiomatization of the programming language PASCAL using this formalism. The primitives of the language are defined by partial correctness formulae which, like those above, state how a language construct will transform a predicate. For example, assignment to a simple variable is defined by

$$\frac{x}{P \ (x \leftarrow E) \ P} \\ E$$

This states that if P holds after x is assigned E , then P with every occurrence of x replaced by an occurrence of E is true before the assignment. Hoare's techniques were taken further in [Dijkstra, 1975, 1976] where Hoare's partial correctness formulae are extended to Dijkstra's predicate transformers. Where Hoare would write $P \{A\} R$, Dijkstra would write $P = wp(A, R)$, indicating that P is the weakest predicate which guarantees both that A terminates and that R will hold afterwards. Thus, Dijkstra's predicate transformers strengthen Hoare's work to deal with total correctness. Dijkstra also used his notions to define a language with limited non-determinism. Another extension called the *intermittent assertion method* [Manna & Waldinger, 1976], also allows proofs of total correctness; it uses assertions which are not invariants but which must hold at least once (hence the name intermittent assertion). [Pratt, 1976] presents foundational work providing a semantic model for these formalisms and a logic in which the methods of Floyd, Hoare, Dijkstra, etc. can be compared. A host of literature analyzing the theoretical and computational foundations of these methods has appeared in recent years such as [Lipton, 1977] and [Jones & Muchnick, 1977] to chose two at random.

Floyd's method was developed for flow chart programs and uses inductive techniques implicitly. [Manna & Pnueli, 1970] generalized Floyd's techniques to handle recursive programs as well. Inductive arguments are relied on. Other inductive techniques were also developed, including computational induction [Park, 1969], recursion induction [McCarthy, 1963] and structural induction [Burstall, 1969]; sub-goal induction, a variant of the inductive assertion method, is presented in [Morris & Wegbreit, 1977]. Structural induction "inducts" on the depth of recursion of the program's data structures; computational and recursion induction are inductions on the depth of function calling. Floyd's inductive assertion method is an induction on the length of the computation path. These methods are surveyed in [Manna, Ness & Vuillemin, 1972] and also in [Reynolds, & Yeh, 1976]. [Manna, 1974] covers a wide range of theoretical issues underlying program verification and related fields.

Following Floyd and Hoare's seminal papers a literature began to develop in which various hand proofs of program correctness were presented. These include [Hoare, 1971] and [London, 1970a,b,c 1971]. Attempts to automate the process soon followed. The first of these was [King, 1969], a very fast verifier of limited power. King's system was coded in assembly language and had built in several special purpose features for simplifying expressions and for handling systems of linear inequalities.

The second system within the Floyd-Hoare framework was PIVOT, implemented by Peter Deutsch [Deutsch, 1973]. Pivot verified programs written in a limited Algol-like language; it works in a manner more similar to my symbolic interpreter than do many of the later systems. PIVOT traversed the program text in forward order (i.e. it started at the beginning and moved towards the end) and interleaved simplification of expressions with interpretation of the program text. It also used a context mechanism to record the values of variables and the truth value of clauses. The context mechanism allowed PIVOT to have an incremental view of the computation's temporal progression. PIVOT had a fixed sequence of deductive techniques which it employed repeatedly. It worked by refutation, trying to reduce the negation of the goal to a contradiction. It was, thus, more like a resolution theorem prover than many of the other verification systems since.

Three further systems followed, inspired largely by the original implementation of the Stanford Verifier [Igarashi, et. al. 1973]. Igarashi, London and Luckham reduced Hoare's logical system to a core of rules which were deduction complete (i.e. anything the full set could deduce, the core could as well) and which, furthermore, could be used deterministically. The Stanford group's set of rules was chosen so that there

For Complex Program Understanding

would always be exactly one rule which could be applied at a time. These rules were used in a backwards manner to create a series of subgoals for the output assertion. For example, if the program were:

$$P(A \dots B; x \leftarrow E) Q$$

there would be exactly one rule whose consequent matches the expression within the braces. For example:

$$\begin{array}{c} x \\ P(A \dots B) Q \\ E \\ \hline P(A \dots B; x \leftarrow E) Q \end{array}$$

These rules are applied repeatedly until the inside of the braces contains an empty program. An implication made from the two formulae surrounding the empty braces is handed to a theorem prover; if the implication can be proven, the program is correct. Originally, the Stanford system used a resolution theorem prover [Allen & Luckham, 1970], but this was replaced by an algebraic and logical reduction system implemented by Suzuki [Suzuki, 1975]. Further work on the Stanford verifier includes a technique for proving termination [Luckham & Suzuki, 1975] which inserts in each loop a "ghost" variable to count the number of repetitions. Termination is proved by demonstrating an upper bound for the ghost variable. [Luckham & Suzuki, 1976] extended the proof rules to include more complex data structures including records, arrays and pointers. [Nelson & Oppen, 1978] have added a more powerful and efficient simplifier to the Stanford system.

A second verification system was started at Stanford Research Institute [Elspas, Levitt, & Waldinger, 1973] which used a verification condition generator similar to that of Igarashi, et. al. However, the SRI system was built around a natural deduction theorem prover of some power, implemented in the language QA4. QA4 has a very powerful collection of data types including sets, bags, and tuples built into the language which allow the theorem prover to ignore issues like the canonicalization of arithmetic expressions. QA4 also provides contexts and backtracking facilities for hypothetical reasoning. (Although Doyle's TMS makes these facilities unnecessary in REASON, at the time of their incorporation into QA4, they represented a clear step forward in theorem proving languages). By using QA4, the SRI group was able to build the theorem prover as a set of small QA4 procedures, grouped into clusters.

The SRI system is also reported on in [Waldinger & Levitt, 1974].

A third system was developed jointly between the Information Sciences Institute at USC and the Automatic Theorem Prover project at the University of Texas [Good, London & Bledsoe, 1975]. This system uses a modified version of the Stanford verification condition generator, a powerful logical and algebraic manipulation package called REDUCE [Hearn, 1971] and the theorem prover of [Bledsoe & Bruell, 1973]. The theorem prover is a natural deduction system with special heuristics for case splitting, interval arithmetic (based on the technique of [Bundy, 1973]), and range splitting in quantified statements [Bledsoe, 1971]. The verifier itself was proven correct (by hand) in [Ragland, 1973].

Two other verification systems of note have been developed; neither of these uses the Floyd-Hoare framework. The Boyer-Moore theorem prover for recursive function theory [Boyer & Moore, 1975,77] uses structural induction rather than inductive assertions; it states both the program and its specifications in Pure LISP, using symbolic execution to reduce the expression. Their system contains powerful methods for choosing the basis for an induction and for generalizing sub-goals into lemmas. The system has proved impressive theorems in recursive function theory; it has also verified a fast string searching algorithm and an arithmetic simplifier.

The other verification system, [Milner, 1972a,b], [Milner & Weyrauch, 1972], is a proof checker for Scott's Logic of Computable Functions (LCF) [Scott, 1972]. The strength of the LCF system is that it operates within a powerful formal logic within which it is possible to reason about complex procedures which manipulate procedures as objects. Lisp programs which we would find difficult to handle within our system are handled directly within LCF. The system was implemented as a proof checker to assist the human proof constructor. VonHenke has done some work on integrating abstract recursive structures in the system [vonHenke, 1975]. In later work, [Gordon, Milner, et al., 1978], the system was extended to facilitate the semi-automatic generation of proofs and the integration of new strategies and types.

Of the systems I have mentioned so far, the one most similar to REASON is Peter Deutsch's PIVOT. Both systems use symbolic interpretation in a forward direction and interleave simplification with evaluation. The other Floyd-Hoare systems use verification condition generators which reduce the entire computation history to a single first order implication. A symbolic evaluation system quite similar to REASON and Deutsch's PIVOT is described in [Hantler & King, 1976]. Other symbolic

execution systems have been used in a program testing (as opposed to verification) environment to form symbolic expressions for the values of program variables. Typical of these is [King, 1976], [Clarke, 1976] and [Howden, 1977,78]. [Balzer, 1978] uses a weak form of symbolic evaluation to fill in the omitted details of an imprecise program specification. [Yonezawa, 1977] describes a system quite similar to mine which is based on that in [Hewitt & Smith, 1975]. However, his system was never implemented.

REASON views programs more dynamically than do many of these other systems. During its normal reasoning it moves assertions backward and forward through the situations, reflecting the dynamic propagation of facts through the situations of the program. In addition, when used within the recognition system, REASON takes an even more dynamic view, expanding the program's temporal behavior and resegmenting this into logical units. This process oriented view more closely resembles the recent theoretical of Pratt on process logic [Pratt, 1978] and the work of Pnueli [Pnueli, 1977] on temporal logics suitable for describing non-terminating programs like operating systems.

Section 15.1: Newer Areas of Verification Research

Synthesizing Loop Invariants

There are several areas of current research on verification systems which I would like to mention before going on to program understanding research more similar to my own. The first of these is the problem of synthesizing loop invariants. The Boyer-Moore system does not need to form loop invariants, but rather uses structural induction on the data structures to achieve the same effect. It has a number of heuristics for choosing which data structure to "induct" on, and works completely automatically. The Floyd-Hoare systems, however, require some other process, human or machine, to generate the inductive assertions. Although, [Dijkstra, 1976] argues that human programmers ought to generate loop assertions as the first step of designing their programs, many researchers have found this cumbersome and would prefer to have an automated loop invariant synthesizer.

Early research in this area includes [Cooper, 1971], [Elspas, 1974] [Elspas, et. al., 1972] in which recurrence relations are generated through use of a ghost loop counter. [Wegbreit, 1973,74] introduced a number of heuristic techniques which involve strengthening and weakening of assertions. Typically the original assertion is one known to hold immediately after the loop's exit (this is easily calculated by the normal VCG procedures) or immediately before entrance to the loop. Strengthening heuristics include dropping a clause from a disjunction or adding one to a conjunction. Wegbreit's techniques were implemented in [German, 1974] and reported on in [German & Wegbreit, 1975]. [Katz & Manna, 1973,76] have developed similar techniques including some for handling arrays and for strengthening a partially correct loop invariant. [Greif & Waldinger, 1974] have also studied some techniques in this area.

More recent work in automatic synthesis of invariants have included [Caplain, 1975], implementation efforts such as those of [Dershowitz & Manna, 1977] and [Moriconi, 1974]. [Cousot & Halbwachs, 1978] presents a method for automatically deriving linear inequalities among the variables of loop; these inequalities may be used as invariants. [Basu & Misra, 1976] have studied some classes of programs such as accumulation loops in which the loop invariant is especially easy to develop automatically. Their work resembles to a small degree [Waters, 1978] work on loop analysis, in that both look for standard patterns within a loop and build a description of the loop based upon known properties of these standard patterns.

For Complex Program Understanding

REASON currently relies exclusively on Waters' techniques plus more advanced recognition techniques being worked on in [Rich, 1978]. I do not yet know whether this will be completely sufficient, or whether I will have to include some of the heuristic techniques mentioned above. Probably, REASON will only use such techniques as a last resort strategy.

Abstraction Techniques

Another major area of ongoing research is the development of abstraction techniques which allow the program and its proof to be structured into layers of smaller procedures. Various approaches have been taken. [Hoare & Wirth, 1973] in their axiomatization of PASCAL include a procedure call rule which is the basis for any form of procedural abstraction. Various technical difficulties with the rule have been discussed in [Cartwright & Oppen, 1978] and [Guttag, et. al., 1977]. However the ability to refer to a procedure by its specification is only the first step in most abstraction techniques.

Frequently, abstraction techniques have been concerned with specifying, implementing, and proving the correctness of abstract data structures. [Hoare, 1972] introduces a method for proving the correctness of a data structure implementation, using the notion of an abstraction function to map between the variable of the concrete space and the variables of the abstract space. [Parnas, 1972] develops a method for hierarchically specifying a system in which each level of procedure is built from modules at a lower level. Modules consist of two types of procedures: O procedures which are allowed to have side effects and V procedures which cannot. Thus, the values of the various V functions characterize the module's state, and the O procedures can be specified in terms of their effect on the values of the V functions. This is quite similar to our method of specifying side effects and has been used in [Robinson & Levitt, 1977].

Some newer languages such as Alphard [Wulf, et. al., 1976] and CLU [Liskov, et. al., 1977] have extensive facilities for grouping procedures together into a "data abstraction" with each procedure representing some of the behavioral capabilities of the abstract datum. The procedures of the data abstraction share access to the concrete representation, while procedures outside are, in general, denied such access. Motivated by these languages, a specification technique, called data algebras, has been developed in [Guttag, 1975] and [Zilles, 1975]. In this technique the data abstraction is specified by axioms containing equations interrelating the behavior of the functions

contained in the data abstraction. Verification consists of showing that each module in the "cluster" preserves each of the axioms. An inductive argument then shows that the axioms are invariants of the cluster since only the modules of the cluster may act on objects of the abstract type. This approach is quite different than those we use; [Yonezawa, 1977] presents an argument for techniques more similar to ours. [Liskov & Berzins, 1977] have written a survey of data specification techniques.

Side Effect on Complex Data Structures

A closely related area of research has dealt with the problems of side effects on complex and shared data structures. [Oppen, 1975] presents an axiom system for reasoning about Directed Graphs and derived a computability result for this system. [Yellowitz & Duncan, 1978] also work with the DiGraph model, but develop a much more succinct formalism.

[Suzuki, 1975] develops axioms for the complex data structures allowed in PASCAL. In this system each data type is made to appear to be an array, indexed by pointers of the appropriate data type. This is possible since in PASCAL a pointer variable may only reference objects of a single type. Suzuki requires a special notation for predicates which refers to a recursive data type; they must include symbols to refer to all the data types involved in the definition. Supposing one wished to have a predicate stating a well formedness criterion for lists of pairs. This would have to be stated as

(Well-formed List-of-Pair-1 P@list P@Pair)

where the last two symbols represent the pseudo-arrays of lists and pairs. If a side effect is performed on any list or pair, this predicate will be updated by changing one of the last two symbols in a manner analogous to the standard array rule. Although logically sound, the system produces large expressions whose intuitive meaning is, at best, unclear. In large interrelated systems, the expressions might well become intractable. Suzuki's technique can be viewed as a special case of my potential-dependency network, in which a very coarse filter is used to decide if an assertion is threatened by a side effect. Whereas REASON filters for specific types of side effects to a particular data type, Suzuki's system filters for any side effect; thus, his system will produce unduly complicated expressions.

[Yonezawa, 1977] uses a formalism which, except for the use of the TMS, is quite a bit like mine; this is hardly accidental since we shared an office for 2 years and both worked with Carl Hewitt. His thesis describes a system which uses the situational calculus to reason about the situational transformation brought about by a side effect. However, Yonezawa's later interests moved more towards a formalism for reasoning about parallel procedures and synchronization; his system was never implemented.

Section 15.2: Apprentice-Like Systems

There has been some other research which includes one or another of the characteristics of the programmer's apprentice project. I will briefly review two categories of these: The first involves some attempt to support the programmer in an interactive design and evolution environment; The second involves some attempt to catalogue standard programming knowledge.

Systems for Evolutionary Design

Most developed among the systems which support interactive design and modification is the system of [Moriconi, 1977]. This system, called SID, is integrated into the University of Texas/ISI verification system of [Good, London & Bledsoe, 1975]. The new feature of this system is an interface module which analyzes proofs constructed by the theorem prover to determine the dependencies between the verification conditions. These are then represented in a simple network. The verification conditions are also analyzed so that their dependence on particular lines of code is recorded in the data base. When program modifications are proposed, the network is examined to see whether any logical link is affected. This is quite similar to the apprentice's purpose links. However, if some dependency is affected, both the verification condition generator and the theorem prover must be called to completely recreate the proof for the modified section of code. Thus, the system is less incremental in its analysis than is REASON which uses the Truth Maintenance System to reuse as much of the original reasoning as possible.

Another project of a similar nature is that of [Dershowitz & Manna, 1977] and [Katz & Manna, 1976]. In the first of these, analogies between the new and old versions of the program specs are used to generate loop invariants for the new code. In the second, a table of dependencies between loop invariants is maintained to aid in the analysis of modifications. Both of these system seem less developed than Moriconi's. The Dershowitz and Manna paper has a brief discussion of the use of schemata to capture some programming generalities, but none of these systems have our emphasis on cataloging programming knowledge.

For Complex Program Understanding

Knowledge Based Systems

Several systems have, however, attempted to take a knowledge based approach. [Schwartz, 1977] proposes to build a set of root programs representing basic programming techniques and a set of correctness preserving combination rules. However, this process is to take place in a strictly hierarchical manner; the work is also intimately tied to the language SETL, in contrast to our attempt to remain language independent.

[Gerhart, 1975a] proposes to catalogue programming knowledge in the form of schemata which serve as syntactic templates. Programming knowledge and proof rules are attached to each template, forming a catalogue of programming knowledge. I feel that this representation is too low level and language dependent, even though some generality is regained through use of correctness preserving transformations [Gerhart, 1975b]. [Darlington & Burstall, 1973] have also studied the use of transformations.

An early attempt to catalogue programming knowledge is found in [Ruth, 1973,76a]. Ruth represented algorithms as grammar's for a parser using an ATN-like formalism. Each grammar can parse several programs representing different implementations (including some with standard bugs) of the same algorithm. The system was developed to parse the programs of beginning students and lacks several features which seem important in the more complex domains which concern me. The ATN-like formalism seems overly syntactic and cumbersome for the representation of the wide range of programming knowledge which I desire to capture. Also, since the formalism has no means of stating the intrinsic behavior of sub-segments, it has no ability to represent the purposeful nature of the interconnections between modules. Later work, [Ruth, 1976b,c] develops an expert system for inventory data bases with extensive knowledge of file and record organization.

The knowledge based system most similar in approach to the apprentice system is the PSI system of [Green, et. al., 1976,77] (in particular the PECOS sub-system developed in [Barstow, 77]). PECOS is a set of refinement rules for program synthesis representing a broad range of knowledge about sets, mappings, tuples, arrays, etc. arranged to capture as much general knowledge as possible. Although the system has some implementation and representational problems (it has neither a deductive system, nor a clear notion of data flow), it does seem to capture a reasonable segment of the knowledge of the expert programmer in a natural manner. Efficiency knowledge is

represented in another sub-system called LIBRA, [Kant, 1977]. The entire PSI system has been capable of synthesizing the code for a simple version of a learning program.

A different type of synthesis system has been developed by [Manna & Waldinger, 1977] in which far more reliance is placed on deductive capabilities. Their system is capable of synthesizing a program to satisfy a given set of input-output specifications. It has rules for loop, recursion, and test formation as well as a method for handling destructive interference between simultaneous sub-goals. Another recent knowledge based system is the SAFE system of [Balzer, et. al., 1977] which takes informal program specifications and attempts to translate these into a precise formal description from which a program may be synthesized.

An excellent treatment of many aspects of program verification and synthesis can be found in [Manna & Waldinger, 1978].

Section 15.3: Dependency Based Reasoning

This thesis has been interlaced with many references to the works of my close colleagues at the MIT AI lab. My approach to developing the deductive machinery of REASON has been influenced heavily by the AI Lab's intellectual atmosphere. Dependency based systems like [Stallman & Sussman, 1977] and [Doyle, 1978] have strongly influenced the redesign of REASON. Another dependency based system which has influence my thought is [London, 1977]. [Doyle, 78] surveys much of the current literature on dependency based systems.

The idea of explicit representation of control as a stepping stone to introspective systems is also heavily influenced by [Doyle, 1978] as well as by [DeKleer, et. al., 1977]. [McDermott, 1976] introduced the task network formalism which he developed considerably further in his NASL system than I have yet done in REASON. The idea of this form of organization was called to my attention by Doyle (private communication). [Davis 1976] uses a weak form of self reflection in a backward chaining system.

A strong influence on my approach to handling side effects has been the considerable AI literature on the frame problem. [McCarthy & Hayes, 1969] and [Hayes, 1971a,b] are good introductions to this issue. [Raphael, 1970] surveys the known techniques for handling the problem.

REASON's entire design is shaped by the central importance given to plans within the apprentice system. HACKER [Sussman, 1975] introduced many of the ideas which helped us develop the plan formalism. Other ideas such as the categorization of plans into particular types first appeared in [Goldstein, 1974]. The plan formalism as we now use it first appeared in [Rich & Shrobe, 1976]. Its current form was heavily influenced by Waters work in [Waters, 1977]. [Sacerdoti, 1975,75a] developed a similar formalism as part of a plan compilation system. Johan deKleer's [deKleer, 1976,77] work on understanding electronic circuits and Allen Brown's [Brown, 1977] work on isolating failures in a circuit have also influence the plan formalism.

[Miller & Goldstein, 1976a,b,77] have been developing a notion of plan for use in tutoring sessions. They have catalogued very general problem solving strategies, whose plans they represent in an ATN-like grammar. The grammar is used to parse the protocol of a student's coding sessions, so that the computer tutor can provide advice. A similar methodology is used in [Genesereth, 1978] to help experts using the MACSYMA symbolic manipulation system. However, in this case the assumption is that the expert has correctly formulated a plan, but has based his plan on faulty knowledge of the MACSYMA facilities. Genesereth's system functions as a MACSYMA consultant, not a tutor.

The earliest mention of a system like the programmer's apprentice is in [Floyd, 1971], although verification and related techniques were not yet well enough developed to do much with the proposal. The pressures of engineering large Artificial Intelligence systems led to another exploration of the idea in [Winograd, 1973]. [Hewitt & Smith, 1975] developed the idea further within the framework of the PLASMA programming language. Both Hewitt and Smith encouraged me and provided some initial insights.

I began this thesis by placing it within the context of a developing set of techniques which I hope will help develop truly self-conscious systems. This document is not intended to answer many of the difficult problems which will lie on that course, but only to develop some technical foundations in program understanding which will help those more bold than I. Minsky [Minsky, 1968] and McCarthy's [McCarthy, 1968] ideas of a decade ago, still lie ahead of us, waiting for solution.

Bibliography

- Balzer, R. 1973 "Automatic Programming", Institute Technical Memo, University of Southern California / Information Sciences Institute, Los Angeles, Cal.
- Balzer, R. et. al. 1974 *Domain-Independent Automatic Programming*, ISI/RR-73-14 University of Southern California (March 1974).
- Balzer, R; Goldman, N; Wile, D. 1978, "Meta-Evaluation as a Tool for Program Understanding", ISI/RR-78-69, January 1978.
- Barstow, David 1977, *Automatic Construction of Algorithms and Data Structures*, PhD. Thesis, Stanford University, September 1977.
- Barstow, David and Kant, Elaine 1976, "Observations on The Interaction of Coding and Efficiency Knowledge in the PSI Program Synthesis System", *Proceedings of The Second International Conference on Software Engineering*, San Francisco, California, October 1976, pp. 19 - 31.
- Basu, S. & Misra, J. 1975, "Proving Loop Programs", *IEEE Trans. on Software Engineering* Vol. 1 Number 1, March 1975.
- Basu, S. & Misra, J. 1976, "Some Classes of Naturally Provable Programs", *Second International Conference on Software Engineering*, pp. 400-406, Oct. 1976.
- Bauer, M. 1975 "A Basis for the Acquisition of Procedures from Protocols", *Proceedings of The Fourth International Joint Conference on Artificial Intelligence*, Tbilisi, Georgia U.S.S.R. September 1975.
- Birtwistle, G.M.; Dahl, Ole-Johan; Myhrhaug, B.; and Nygaard, K. 1973 *SIMULA BEGIN* Auerbach. 1973
- Bledsoe, W.W. 1971, "Splitting and Reduction Heuristics in Automatic Theorem Proving", *Artificial Intelligence*, vol. 2, pp. 55-77, North Holland Publishing Co., Amsterdam, 1971.

- Bledsoe, W.W. & Bruell, P. 1973 "A Man-Machine Theorem Proving System", *Advance Papers of The Third International Joint Conference on Artificial Intelligence* Stanford University, August 1973 pp. 55-65; also *Artificial Intelligence* vol 5. no. 1 pp. 51-72, Spring 1974.
- Bose, A. and Stevens, K 1965, *Introductory Network Theory*, Harper and Row, New York, 1965, page 1.
- Boyer, R.S. & Moore, J.S. 1975 "Proving Theorems About LISP Functions", *Journal of the Association for Computing Machinery* vol. 22 no. 1, January 1975.
- Boyer, R.S. & Moore, J.S. 1977, "A Lemma Driven Automatic Theorem Prover for Recursive Function Theory", *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* Cambridge Mass. pp. 511-519, August 1977.
- Brown, A.L. 1977 *Qualitative Knowledge, Causal Reasoning, and the Localization of Failures*, M.I.T. Artificial Intelligence Laboratory Technical Report 362, March 1977.
- Bundy, A. 1973, "Doing Arithmetic With Diagrams", *Advance papers of the Third International Joint Conference on Artificial Intelligence* Stanford University pp. 130-138, August 1973.
- Burstall, R. M. 1974, "Program Proving As Hand Simulation and A Little Induction", *Proceedings of IFIP Conference 1974*.
- Burstall, R. M. 1969 "Proving Properties of Programs by Structural Induction", *Computer Journal* vol. 12, pp. 4-8
- Burstall, R. M. 1972 "Some Techniques for Proving Properties of Programs Which Alter Data Structures", *Machine Intelligence 7*, Edinburgh University Press.
- Caplain, M. 1975, "Finding Invariant Assertions for Proving Programs", *Proc. International Conference on Reliable Software*, Los Angeles Calif. April 1975.

- Cartwright, R. and Oppen, D. 1978, "Unrestricted Procedure Calls in Hoare's Logic", *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, Tucson, Arizona, Jan. 1978. pp. 131-140.
- Clarke, Lori A. 1976, "A System to Generate Test Data and Symbolically Execute Programs", *IEEE Transactions on Software Engineering* vol. SE-2 no. 3 September 1976 pp. 215-222.
- Cooper, D.C. 1971, "Programs for Mechanical Program Verification" *Machine Intelligence 6*, American Elsvier, New York, 1971 pp. 3-59.
- Cousot, P and Halbwachs, N. 1978, "Automatic Discovery of Linear Restraints Among Variables of a Program", *Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages*, pp. 84-96, Jan. 1978.
- Dahl, O.J., Dijkstra, E., And Hoare, C. A. R. 1972 *Structured Programming*, Academic Press, 1972.
- Darlington, J. and Burstall, R.M. 1973 "A System Which Automatically Improves Programs", *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.
- Darlington, Jared L. 1973a "Automatic Program Synthesis in Second-Order Logic", *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, August 1973.
- Davis, R. 1976, "Application of Meta-Level Knowledge to The Construction, Maintenance, and Use of a Large Knowledge Base", Stanford AIM-238, 1976.
- DeMillo, R.A.;Lipton, R. and Perlis, A. 1977, "Social Processes in Proving Theorems and Programs", *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, 1978.
- Dershowitz, N. and Manna Z. 1974, "The Evolution of Programs: Automatic Program Modification", *IEEE Transactions on Software Engineering*, vol SE-3 no. 6. Nov. 1977, pp. 377-385.

AD-A078 055

MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTE--ETC F/G 6/4
DEPENDENCY DIRECTED REASONING FOR COMPLEX PROGRAM UNDERSTANDING--ETC(U)
APR 79 H E SHROBE
N00014-75-C-0643

UNCLASSIFIED

AI-TR-503

NL

4 OF 4
AD-
A078055



END
DATE
FILMED

1-80
DDC

- Dershowitz, N. and Manna, Z. 1974, "Inference Rules for Program Annotation", Stanform AIM-303, October 1977.
- Deutsch, L.P. 1973, *An Interactive Program Verifier*, PhD. Thesis University of California at Berkeley, June 1973.
- Dijkstra, E.W. 1975, "Guarded commands, nondeterminacy and formal derivation of programs", *Communications of the ACM*, vol 18, pp. 453-457, Aug. 1975.
- Dijkstra, E.W. 1976, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. 1976
- DeKleer, Johan 1976, "Local Methods for Localization of Faults in Electronic Circuits", M.I.T. Artificial Intelligence Laboratory Memo 394.
- DeKleer, Johan 1977, "A Theory of Plans for Electronic Circuits", MIT Artificial Intelligence Laboratory Working Paper 144, May 1977.
- DeKleer, J., Doyle, J., Steele, G. & Sussman, G.J. "AMORD: Explicit Control of Reasoning", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester, August 1977.
- Dolotta, T.A. and Mashey, J.R. 1976, "An Introduction to the Programmer's Workbench", *Proceedings of the Second International Conference on Software Engineering*, San Francisco, Cal., October 1976, pp. 164-168.
- Donzea-Gouge, V., Huet G., Kahn, G., Lang, B., and Levy, J.J. 1975 "A Structure-Oriented Program Editor: A First Step Towards Computer Assisted Programming", Report 114, Institut de Recherche en Informatique et Automatique, France.
- Doyle, Jon 1977, "More Explicit Control of Reasoning", unpublished manuscript.
- Doyle, Jon 1978 *Truth Maintenance Systems for Problem Solving*, M.I.T. Artificial Intelligence Laboratory Technical Report 419, January 1978.

- Doyle, Jon 1978b, "Reflexive Interpreters", Proposal for Research Leading to the degree of Ph.D. M.I.T. Dept. of EE&CS, June 1978.
- Elsapas, Bernard, 1974, "The Semi-Automatic Generation of Inductive Assertions for Proving Program Correctness", SRI Project 2686, July 1974.
- Elsapas, B.; Green, M.; Levitt, K. and Waldinger, R. 1972, *Research in Interactive Program-Proving Techniques*, SRI Project 8398 Phase II Report, May 1972.
- Elsapas, B., Levitt, K.N., & Waldinger, R.J., *An Interactive system for The Verification of Computer Programs*, Stanford Research Institute Project 1891 Final Report, September 1973.
- Fikes, Richard and Nilsson, Nils 1971, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving". *Proceedings of the Second International Joint Conference on Artificial Intelligence* page 608.
- Floyd, R. W. 1967 "Assigning Meaning to Programs", in *Mathematical Aspects of Computer Science*, J.T. Schwartz (ed.) vol. 19, Am. Math. Soc. pp. 19-32. Providence Rhode Island.
- Floyd, R.W. 1971 "Toward Interactive Design of Correct Programs", *IFIP*, 1971.
- Genesereth, Michael 1978, *Automated Consultation for Complex Computer Systems*, PhD. thesis, The Division of Applied Sciences, Harvard University, May 1978.
- Gerhart, S. L. 1975a, "Knowledge About Programs: A Model and Case Study", *SIGPLAN Notices*, Vol. 10, Num. 6, *Proceedings of the International Conference on Reliable Software*.
- Gerhart, S.L. 1975b "Correctness-Preserving Program Transformations", *Proceedings of the Second Symposium on Principles of Programming Languages*, Palo Alto.
- German, S. 1974, *A Program Verifier That Generates Inductive Assertions*, Center For Research in Computing Technology, Harvard University, Cambridge Mass. Tech. Report TR-19-74, Aug. 1974.

- German, S. & Wegbreit, B. 1975, "A Synthesizer of Inductive Assertions", *IEEE Transactions on Software Engineering* vol. 1 num. 1, March 1975.
- Goldstein, Ira 1974, *Understanding Simple Picture Programs*, MIT Artificial Intelligence Laboratory Technical Report 294, September 1974
- Goldstein, Ira 1976, "The Computer As Coach", MIT Artificial Intelligence Laboratory Memo 389, December 1976.
- Goldstein, I.P. and Miller, M.L. 1976 "Structured Planning and Debugging, A Linguistic Theory of Design", MIT AI Lab Memo 387. December, 1976.
- Good, Donald; London, Ralph; & Bledsoe W.W. "An Interactive Program Verification System", *IEEE Transactions on Software Engineering*, vol. SE-1, number 1, March 1975.
- Gordon, M.; Milner, R.; Morris, L.; Newey, M.; and Wadworth, C. 1978, "A MetaLanguage for Interactive Proof in LCF", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson Arizona, Jan. 1978.
- Green, G.C. 1976, "The Design of The PSI Program Synthesis System", *Proceedings of The Second International Conference on Software Engineering*, San Francisco, October 1976, pp. 4 - 18.
- Green, G.C. 1977, "A Summary of The PSI Program Synthesis System", *Proceedings of The Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977, pp. 380 - 381.
- Green, G.C. & Barstow, D.R. "Some Rules for the Automatic Synthesis of Programs", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence* Tbilisi, Georgia, USSR, September 1975.
- Greif, I. and Waldinger R. 1974, "A More Mechanical Heuristic Approach to Program Verification", *Proceeding of International Symposium on Programming*, Paris, April 1974, pp. 83-90.

- Hoare, C.A.R. and Wirth, N. 1973 "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica*, 2,4, pp. 335-355.
- Howden, William E. 1977, "Symbolic Testing and the DISSECT Symbolic Evaluation System", *IEEE Transactions on Software Engineering*, vol SE-3 no. 4, July 1977 pp.266-278.
- Howden, William E. 1978, "DISSECT - A Symbolic Evaluation and Program Testing System", *IEEE Transactions on Software Engineering* vol SE-4 no. 1 January 1978.
- Igarashi S., London R., and Luckham D. 1973, *Automatic Program Verification I: A Logical Basis and Its Implementation*, Stanford AIM-200, May 1973.
- Jones, N. and Muchnik S. 1978, "Even Simple Programs Are Hard To Analyze", *Journal of the ACM* vol 24, no. 2 April 1977 pp. 338-350.
- Kant, E. 1977, "The Selection of Efficient Implementations for A High Level Language", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester August 1977.
- Katz, S.M., and Manna, Z. 1973, "A Heuristic Approach to Program Verification", *Third International Joint Conference on Artificial Intelligence*, Stanford U. September 1973.
- Katz, S. & Manna, Z. 1976, "Logical Analysis of Programs", *Communications of the ACM*, Vol. 19 Num. 4 pp.188-206 April 1976.
- King, J. 1969, *A Program Verifier*, Carnegie Mellon University, 1969.
- King, J.C. 1971 "Proving Programs to be Correct", *IEEE Transactions on Computers*, C-20, 11, Nov. 1971.
- King, J.C. 1976 "Symbolic Execution and Program Testing", *Communications of the ACM*, July, Vol. 19, No. 7, p. 385.

- Hoare, C.A.R. and Wirth, N. 1973 "An Axiomatic Definition of the Programming Language PASCAL", *Acta Informatica*, 2,4, pp. 335-355.
- Howden, William E. 1977, "Symbolic Testing and the DISSECT Symbolic Evaluation System", *IEEE Transactions on Software Engineering*, vol SE-3 no. 4, July 1977 pp.266-278.
- Howden, William E. 1978, "DISSECT - A Symbolic Evaluation and Program Testing System", *IEEE Transactions on Software Engineering* vol SE-4 no. 1 January 1978.
- Igarashi S., London R., and Luckham D. 1973, *Automatic Program Verification I: A Logical Basis and Its Implementation*, Stanford AIM-200, May 1973.
- Jones, N. and Muchnik S. 1978, "Even Simple Programs Are Hard To Analyze", *Journal of the ACM* vol 24, no. 2 April 1977 pp. 338-350.
- Kant, E. 1977, "The Selection of Efficient Implementations for A High Level Language", *Proceedings of the Symposium on Artificial Intelligence and Programming Languages*, University of Rochester August 1977.
- Katz, S.M., and Manna, Z. 1973, "A Heuristic Approach to Program Verification", *Third International Joint Conference on Artificial Intelligence*, Stanford U. September 1973.
- Katz, S. & Manna, Z. 1976, "Logical Analysis of Programs", *Communications of the ACM*, Vol. 19 Num. 4 pp.188-206 April 1976.
- King, J. 1969, *A Program Verifier*, Carnegie Mellon University, 1969.
- King, J.C. 1971 "Proving Programs to be Correct", *IEEE Transactions on Computers*, C-20, 11, Nov. 1971.
- King, J.C. 1976 "Symbolic Execution and Program Testing", *Communications of the ACM*, July, Vol. 19, No. 7, p. 385.

- Knuth, D.E. 1968 *The Art of Computer Programming*, Vol. 1, Addison-Wesely.
- Lipton, R.J. 1977, "A Necessary and Sufficient Condition for the Existence of Hoare Logics", XEROX PARC CSL-77-4, June 1977.
- Liskov, B. 1974, "A Note on CLU", MIT/Computation Structures Group Memo 112, MIT/LCS, November 1974.
- Liskov, B.; Snyder, Alan; Atkinson, Russell; and Schaffert, Craig; 1977, "Abstraction Mechanisms in CLU", *Communications of the ACM*, August 1977, pp. 564 - 576.
- Liskov, B. and Berzins, V. "An Appraisal of Program Specifications" M.L.T. Computation Structures Group Memo 141-1. April 1977.
- Liskov, B. and Zilles S. 1974 "Programming with Abstract Data Types", Proc. of Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4.
- Liskov, B. & Zilles, S.N. 1975, "Specification Techniques for Data Abstractions", *IEEE Transactions on Software Engineering*, Vol. SE-1 No. 1, March 1975.
- Litvintchouk, S.D. and Pratt, V.R. 1977, *A Proof-Checker for Dynamic Logic*, MIT AI Memo 429, June 1977.
- London, P. 1977, "A Dependency-Based Modelling Mechanism for Problem Solving", Univ. of Maryland, Computer Science Dept TR-589, Nov. 1977.
- London, R. 1975 "A View of Program Verification", *ACM SIGPLAN Notices*, Vol. 10, No. 6, Proc. of International Conf. on Reliable Software.
- Long, W.J. 1977 "A Program Writer". MIT/LCS/TR-107, November 1977 (Ph.D. Thesis)
- Luckham, David C. & Suzuki, Norihisa, "Automatic Program Verification IV: Proof of Termination Within a Weak Logic of Programs", Stanford AIM-269, October 1975.

- Luckham, David C. & Suzuki, Norihisa 1976, "Automatic Program Verification V: Verification-Oriented Proof Rules for Arrays, Records and Pointers", Stanford AIM-278, March 1976.
- Macsyma 1975, *The MACSYMA reference Manual*, The Mathlab Group, MIT Laboratory for Computer Science, November 1975.
- Manna, Z. 1969, "Properties of Programs and The First-order Predicate Calculus", *Journal of the ACM* vol. 16, no. 2, pp. 244-249 244-255, April 1969.
- Manna, Z. 1969, "The Correctness of Programs", *Journal of Computational Systems Sciences*, vol. 3 no. 2 pp. 119-127, May 1969.
- Manna, Z. and Pnueli, A. 1969, "Formalization of Properties of Recursively Defined Functions", *Proceedings of the ACM Symposium of Theory of Computing* pp. 201-210, ACM New York 1969.
- Manna, Z. and Pnueli, A. 1970, "Formalization of Properties of Functional Programs", *Journal of the ACM* vol. 17 no. 3 pp. 555-569 July 1970.
- Manna, Z. and Pnueli, A. 1974, "Axiomatic Approach to Total Correctness of Programs", *Acta Informatica* 3, pp. 253-263, 1974.
- Manna, Z. and Waldinger, R. 1975, "Knowledge and Reasoning in Program Synthesis", *Artificial Intelligence* 6, pp. 175-208.
- Manna, Z. and Waldinger, R. 1976, "Is 'sometime' sometimes better than 'always'? Intermittent assertions in proving program correctness" *Proceedings of the Second International Conference on Software Engineering*, October 1976.
- Manna, Z. and Waldinger, R. 1977, *Synthesis: Dreams => Programs*, Stanford Research Institute Technical Note 156, November 1977.
- Manna, Z. and Waldinger, R. 1978, "The Logic of Computer Programming", *IEEE Transactions on Software Engineering*, vol SE-4 no. 3, May 1978, pp. 199-229.

- McCarthy, J. 1962a, "Computer Programs for Checking Mathematical Proofs" in *Recursive Function Theory, Proceedings of Symposia in Pure Mathematics*, vol. 5, Am. Math. Soc.
- McCarthy, J. 1962b, *Towards a Mathematical Theory of Computation. Proceedings of 1962 IFIP.*
- McCarthy, J. 1963, *A Basis for a Mathematical Theory of Computation*, in *Computer Programming and Formal Systems*, P. Braffort & D. Hershberg eds. North-Holland, Amsterdam 1963.
- McCarthy, J. 1964, *A Formal Description of a Subset of Algol*, *Proceeding of the conference on Formal Language Description Languages*, Vienna 1964.
- McCarthy J. and Painter J.A. 1966, "Correctness of A Compiler for Arithmetic Expressions", Stanford University Technical Report CS38, April 29, 1966, also in *Proceeding of a Symposium in Applied Mathematics*, vol. 19 -- *Mathematical Aspects of Computer Science*, pp. 33-41, J.T. Schwartz ed. Am. Math. Soc. Providence R.I.
- McCarthy, J. 1963, "Towards a mathematical science of computation" *Proceedings of IFIP Congress 62*, pp. 21-28, Amsterdam: North Holland.
- McCarthy, J. and Hayes, P. 1969 "Some Philosophical Problems from the Standpoint of Artificial Intelligence", *Machine Intelligence 4*, American Elsevier, N.Y.
- McDermott, Drew Vincent 1976, *Flexibility and Efficiency in a Computer Program for Designing Circuits*, MIT PhD. Thesis, September 1976.
- Mikelsons, M. 1975 "Computer Assisted Application Definition", *Proceedings of the Second ACM Symposium on Principles of Programmings Languages*, Palo Alto.
- Miller, M. L. and Goldstein, L.P. 1976a "SPADE: A Grammar Based Editor For Planning and Debugging Programs", MIT AI Lab Memo 386. December 1976.

- Miller, M. L. and Goldstein, I.P. 1977a "Structured Planning and Debugging" *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, MIT, August 1977.
- Miller, M. L. and Goldstein, I.P. 1977b "Problem Solving Grammars as Formal Tools for Intelligent CAI" *ACM77*, October, 1977.
- Milner, R. 1972a, "Logic for Computable Functions: Description of a Machine Implementation", Stanford AIM-169.
- Milner, R. 1972b, "Implementation and Applications of Scott's Logic for Computable Functions", *Proceedings of an ACM Conference on Proving Assertions about Programs*, SIGPLAN Notices vol. 7, no. 1, pp. 1-6.
- Milner, R. and Weyrauch, R. 1972, "Proving Compiler Correctness in a Mechanized Logic", *Machine Intelligence 7* Meltzer and Michie eds. pp. 51-70 Edinburgh University Press, 1972.
- Minsky, Marvin 1961, "Steps Towards Artificial Intelligence", *Proceedings IRE*, Vol. 49, No. 1, 1961.
- Minsky, Marvin 1963, "A LISP Garbage Collector Using Serial Secondary Storage", MIT AI Memo No. 58 (revised) 1963.
- Minsky, Marvin 1968, "Descriptive Languages and Problem Solving" and "Matter, Mind, and Models" in *Semantic Information Processing*, Marvin Minsky ed., The MIT Press, Cambridge Mass. July 1968.
- Minsky, M., "Form and Content in Computer Science", *Journal of the ACM* 17, No. 2, April 1970, pp. 197-215, 1970 ACM Turing Lecture.
- Moore, J.S. 1974, "Introducing PROG into the PURE LISP Theorem Prover", Xerox PARC Report CSL-74-3.
- Moore, Robert Carter 1975, *Reasoning From Incomplete Knowledge In A Procedural Deduction System*, MIT/AI-TR-347 December 1975.

- Moriconi, M. 1974, "Towards the Iterative Synthesis of Assertions", *Res. Report*, University of Texas at Austin, Oct. 1974.
- Moriconi, M. 1977, *A System For Incrementally Designing and Verifying Programs*, ISI/RR-77-65, November 1977; also PhD Thesis University of Texas, 1977.
- Morris, J.H. & Wegbreit, B. 1977, "Subgoal Induction", *Communications of the ACM*, Vol. 20 Num. 4 pp. 209-222, April 1977.
- Naur, P. 1966, "Proofs of Algorithms by General Snapshots", *BIT* vol. 6, pp. 310-316, 1966.
- Nelson, G and Oppen D. 1978, "A Simplifier Based on Efficient Decision Algorithms", *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona Januaray 1978 pp. 141-150.
- Newell, A. Shaw, J.C. and Simon H. 1959, "Report on a General Problem Solving Program", *Proceedings of the International Conference on Information Processing*, Paris, UNESCO House, 1959.
- Oppen, Derek 1975, *On Logic and Program Verification*, University of Toronto Technical Report 82, (also PhD dissertation) April 1975.
- Oppen, Derek 1978, "Reasoning About Recursively Defined Data Structures", *Conference REcord of the Fifth ACM Symposium on Principles of Programming Languages*, Tucson, Arizona Januaray 1978, pp. 151-157.
- Parnas, D.L. 1972, "A Technique for Software Module Specification with Examples", *Communications of the ACM* Vol. 15, No. 5.
- Paterson, M. and Hewitt, C. 1970 "Comparative Schematology", *Conference Record ACM Conference on Concurrent Systems and Parallel Computation* (1970).
- Pnueli, Amir 1977, "The Temporal Logic of Programs", *18th Annual Symposium of Foundations of Computer Science* pp.46-57, Oct. 77.

- Pratt, V. 1976, "Semantical Considerations on Floyd-Hoare Logic", MIT/LCS/TR-168, September 1976; also in *Proceedings of the 17th Annual IEEE Symposium on Foundations of Computer Science*, pp. 109-121, 1976.
- Pratt, V. 1978, "Process Logic" unpublished manuscript, May 1978.
- Ragland, L.C., *A Verified Program Verifier* 1973, PhD. dissertation, University of Texas, Austin, 1973.
- Raphael, B. 1970, "The Frame Problem in Problem-Solving Systems", SRI Technical Note 33, June 1970.
- Rich, C. 1977, Plan Recognition In A Programmer's Apprentice", MIT/AI Working Paper 147, May 1977.
- Rich, C. 1979 forthcoming, "A Representation for Programming Knowledge and Applications to Recognition, Generation, and Cataloguing of Programs", PhD. thesis forthcoming, January 1979.
- Rich C. and Shrobe H. 1976, *An Initial Report On A LISP Programmer's Apprentice*, MIT/AI/TR-354, December 1976.
- Robinson, L and Levitt, K. 1977, *Communications of the ACM* vol. 20, no. 4 April 1977, pp.271-283.
- Ruth, Gregory 1973, *Analysis of Algorithm Implementations* M.I.T. Ph.d. Thesis, Project MAC Technical Report 130.
- Ruth, Gregory 1976a, "Intelligent Program Analysis", *Artificial Intelligence* 7, Spring 1976, pp. 65 - 85.
- Ruth, Gregory 1976b, "Protosystem I: An Automatic Programming System Prototype", TM-72, MIT Laboratory for Computer Science, 1976 (also in the *Proceedings of the 1978 NCC* in abbreviated form).

- Ruth, Gregory. 1976c, "Automatic Design of Data Processing Systems", *ACM Symposium on Principles of Programming Languages*, 1976.
- Sacerdoti, Earl D. 1973, "Planning in a Hierarchy of Abstract Spaces", *Proceedings of the Thrid International Joint Conference on Artificial Intelligence* Stanford University, September 1973, pp. 412-422.
- Sacerdoti, Earl D. 1975, "The Non-Linear Nature of Plans" Stanford Research Institute A.I. Group Technical Note 101.
- Sacerdoti, Earl D. 1975a, "A Structure for Plans and Behavior", SRI Technical Note 109.
- Schwartz, J.T. 1973, *On Programming, An Interim Report on the SETL Project; Installment I: Generalities*, Courant Institute of Mathematical Sciences, New York University, February 1973.
- Schwartz, J.T. 1977, "On Correct Program Technology", in Courant Computer Science Report #12, September 1977.
- Scott, D. 1972 "Lattice Theory, Data Types and Semantics", in *Formal Semantics of Programming Languages*, Rustin ed, Prentice-Hall, p. 65.
- Shaw, D., Swartout, W., and Green, C. 1975 "Inferring LISP Programs from Examples", *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*, Tbilisi Georgia, U.S.S.R August 1975.
- Shrobe, Howard E. 1978, "Plan Verification in A Programmer's Apprentice", M.I.T. Artificial Intelligence Laboratory Working Paper # 158, January 1978.
- Siklossky, L. and Sykes D. 1975 "Automatic Program Synthesis from Example Problems", *Fourth International Joint Conference on Artificial Intelligence* Tbilisi Georgia, U.S.S.R, August 1975.
- Spitzen, J. & Wegbreit, B. 1975, "The Verification and Synthesis of Data Structures", *Acta Informatica* 4, 1975.

- Stallman, Richard and Sussman, G. J. 1977, "Forward Reasoning and Dependency-Directed Backtracking In A System for Computer-Aided Circuit Analysis", *Artificial Intelligence Journal*, October 1977.
- Steele, Guy L. 1977, "Debunking the 'Expensive Procedure Call' Myth", *Proceedings of ACM-77*, October 1977.
- Strong, H.R. 1970, *Translating Recursion Equations into Flow Charts*, Reports RC 2834 (March 1970) and RC 2859 (April 1970), IBM Yorktown Heights.
- Suppes, Patrick 1957, *Introduction to Logic*, Van Nostrand, New York, 1957.
- Sussman, G.J. 1973, *A Computational Model of Skill Acquisition*, M.I.T. Department of Mathematics Ph.D. Thesis; M.I.T. Artificial Intelligence Laboratory Technical Report 297, August 1973; *A Computer Model of Skill Acquisition*, New York, American Elsevier 1975.
- Sussman, G.J. 1977a, "Electrical Design: A Problem for Artificial Intelligence Research", *Proceedings of The Fifth International Joint Conference on Artificial Intelligence*, Cambridge, Massachusetts, August 1977.
- Sussman, G. J. 1977b, "SLICES: At The Boundary Between Analysis and Synthesis", M.I.T. Artificial Intelligence Laboratory Memo 433, July 1977. (Also to appear in *The Proceedings of The IFIP Working Conference on Artificial Intelligence and Pattern Recognition in Computer Aided Design* in 1978.)
- Sussman, G. J. and Stallman Richard 1975, "Heuristic Techniques in Computer Aided Circuit Analysis", *IEEE Transactions on Circuits and Systems*, Vol. CAS-22, No. 11, November 1975.
- Suzuki, N. 1975, "Automatic Program Verification II: Verifying Programs by Algebraic and Logical Reduction", Stanford AIM-255, December 1974.
- Suzuki, N. 1976, "Automatic Verification of Programs with Complex Data Structures", Stanford AIM-279, February 1976.

- Teitelman, W. et al. 1975, *Interlisp Reference Manual*, Xerox Palo Alto Research Center, Dec. 1975.
- Teitelman, W. 1977 "A Display Oriented Programmer's Assistant" *Proceedings of the Fifth International Joint Conference on Artificial Intelligence*, MIT, August 1977.
- von Henke, F. 1975, "On the Representation of Data Structures in LCF with Applications to Program Generation", Stanford AIM-267, September 1975.
- Von Neumann, J., Goldtine "Planning and Coding Problems for an Electronic Computer Instrument, part 2 vol 1-3", *John von Neumann Collected Works* vol 5, pp. 80-235 (Pergamon Press, New York, 1963).
- Waldinger, R. and Levitt, K.N. 1974 "Reasoning About Programs", *Artificial Intelligence* 5, pp. 235-316, also SRI Technical Note 86, October 1973.
- Waldinger, Richard 1975, "Achieving Several Goals Simultaneously" Stanford Research Institute A.I. Group Technical Note 107.
- Waters, R.C. 1976, "A System for Understanding Mathematical FORTRAN Programs", M.I.T. Artificial Intelligence Laboratory Memo 368, August 1976.
- Waters, R.C. 1977, "A Method, Based on Plans, for Understanding How a Loop Implements a Computation", M.I.T. Artificial Intelligence Laboratory Working Paper 150, July 1977.
- Waters, R.C. 1978, *A Method For Automatically Analyzing the Logical Structure of Programs*, PhD. Thesis M.I.T. (forthcoming) EE&CS, Sept. 1978.
- Wegbreit, B. 1973, "Heuristic Methods for Mechanically Deriving Inductive Assertions", *Proceedings of the Third International Joint Conference on Artificial Intelligence*, Stanford University, September 1973.
- Wegbreit, B. 1974, "The Synthesis of Loop Predicates", *Communications of The ACM*, Vol. 17, pp. 102-112, Feb. 1974.

- Wegbreit, B. 1976, "Constructive Methods In Program Verification", Xerox Palo Alto Research Center CSL-76-2, July 1976.
- Wilczynski, D. 1975 "A Process Elaboration Formalism for Writing and Analyzing Programs", U. of S. Cal. Information Sciences Inst., ISI/RR-75-35.
- Winograd, Terry 1973 "Breaking the Complexity Barrier Again" *Proceedings of the ACM SIGIR-SIGPLAN Interface Meeting*, Nov. 1973.
- Wulf, W.A. 1974, "ALPHARD: Towards a Language to Support Structured Programming", Carnegie Mellon University Dept. of Comp. Sci., April 1974.
- Wulf, W; London, R; and Shaw, M. 1976, "An Introduction to the Construction and Verification of Alphard Programs" *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, December 1976, pp. 253-265.
- Yellowitz, L and Duncan, A. 1978, "Data Structures and Program Correctness: Bridging The Gap", *Computer Languages* vol 3. pp. 135-142.
- Yonezawa, A. 1975, "Meta-Evaluation of Actors With Side Effects", MIT/AI Working Paper 101, June 1975.
- Yonezawa, Aki 1976a "Meta-Evaluation for Verification and Analysis of Actor Programs", Draft paper, M.I.T. A.I. Lab.
- Yonezawa, A. 1976a, "Symbolic-Evaluation As An Aid To Program Synthesis", MIT/AI Working Paper 124, April 1976.
- Yonezawa, A. 1976b, "Symbolic Evaluation Using Conceptual Representations For Programs With Side-Effects", MIT/AI Memo 399, December 1976.
- Yonezawa, A. 1977, *Verification and Specification Techniques for Parallel Programs based on Message-Passing Semantics*. M.I.T. PhD. December, 1977.
- Zilles, S. 1975, "Abstract Specification for Data Types", IBM Research Laboratory, San Jose California, 1975.